Annual Review in Automatic Programming, Vol. 9, pp. 1–84.
© Pergamon Press Ltd., 1978. Printed in Great Britain. 0066-4138/78/0501-0001 $05.00/0

An Introduction to the Formal Definition of ALGOL 68

ANDREW D. McGETTRICK

Department of Computer Science, University of Strathclyde, GLASGOW, Scotland

The aim of this paper is to provide an introduction or guide to the Revised ALGOL 68 Report [15].

The earlier revised ALGOL 60 Report [12] constituted one of the first attempts at formally defining a programming language. This was a major advance and opened up new areas of interest and research in computer science. But there were some deficiencies in both the language and in its formal definition. In the course of time other attempts were made at improving and correcting the earlier oversights.

Our intention here is to begin by considering the formal definition of ALGOL 60 [12] and to show how these early ideas gradually evolved to produce the method of formally defining ALGOL 68. The discussion will, for the most part, be restricted to the ALGOL movement. But over the years other advances have been made in formally defining languages. The most notable of these is probably the development from McCarthy's LISP [7–11] through to the Vienna Definition Language [5, 6, 16, 17] and beyond. But see also [4].

In discussing formal definition of programming languages one must distinguish between the programming language itself and its method of formal definition. These are quite separate. Yet they tend to be confused since new methods of formal definition often accompany new programming languages. And a formal definition tends to provide a vocabulary of terms by which various constructions become known.

It will be assumed that the reader has a certain knowledge of the programming languages ALGOL 60, ALGOL W and ALGOL 68.

Moreover it would be convenient if he had available a copy of the formal definition of each language. Although it should be possible to proceed without these documents frequent references are made to them.

1. ALGOL 60

ALGOL 60 was defined using essentially just syntax and semantics. The syntax was designed to give the set of admissible combinations of characters. The semantics associated a meaning with syntactically correct sequences of characters.

Apart from syntax and semantics, examples were used to illustrate various ideas. Remarks, usually of a formal nature, were included to allow for discussion of, for example, values and types, subscripts, operators, standard functions, transfer functions, etc.

1.1. SYNTAX

The syntax of ALGOL 60 was expressed via a context-free grammar using the metalanguage employing BNF notation and involving the now familiar characters <, >, |, ::=. In this notation nonterminals are indicated by surrounding a sequence of characters with < and >. Thus one had such nonterminals as

<identifier>, *<integer>*, *<compound statement>*, etc.

The BNF definition of ALGOL 60 was for the most part quite accurate though objections could be levelled at it on the grounds that it was incomplete. For it made use of some nonterminals without defining these properly. In the definition of *<proper string>* for example the nonterminal *<any sequence of basic symbols not containing ' or >* was used. In the definition of commentary use was made of the nonterminal *<any sequence not containing ;>*. Note however that these deficiencies could easily have been avoided by including the appropriate definitions. They do not constitute a fault in the method of definition but rather a fault in the application of the method.

Consider however the syntactic rule

<relational operator>::=<|≤|=|≥|>|≠

The symbols < and > enclose a sequence of characters. Those could justifiably be interpreted as a nonterminal. Ambiguity therefore results from the interplay between the metalanguage and the programming language itself. This problem becomes more pronounced if the vertical bar, as often happens in other

programming languages, is used as a terminal to denote the boolean operator **or** or for operations on strings.

1.2. DEFICIENCIES IN BNF NOTATION

There are deficiencies of a more serious nature that can be levelled at BNF when considered as a means of defining programming languages (see [4]). The trouble stems from the fact that too large a set of syntactically legal but erroneous programs are defined. Some illustrative examples are given below.

The ALGOL 60 assignment statement

$$x := y$$

is legal only if x and y have been suitably declared within the current or an enclosing block. This and similar situations in which a variable can appear only if a suitable declaration has also appeared are not covered by BNF.

Whenever subscripted variables or procedure calls are used it is important that the correct number of subscripts or actual parameters are used. In the case of procedures the number and the types of the actual parameters must match the number and types of the corresponding formal parameters. No such checks are performed in the syntax of the ALGOL 60 Report.

Precisely what constitutes a suitable declaration is not defined by the syntax. A declaration such as

integer x

is legal only if x has not previously been declared within the same block. This applies to declarations of all variables, labels, switches, procedures and functions.

In summary then many of the ALGOL 60 constructs are legal only if they appear in a suitable context, the context being defined by the declarations of identifiers accessible from within the current block. These context sensitive requirements are not contained in the syntax of the ALGOL 60 Report.

1.3. SEMANTICS

The semantics of ALGOL 60 were described for the most part in English. This is where most of the inaccuracies and blemishes in the definition of ALGOL 60 arose.

The **own** concept introduced ambiguities. It was not clear what interpretation should be given to dynamic **own** arrays or to recursive procedures containing declarations of **own** variables. Other trouble spots were the use of side effects, the problem of repeated formal parameters, the effect of a jump from the body of a function designator, etc. These are all discussed in detail in [3].

But besides English there were some places where another technique was also used. Consider (see [12] section 4.5.3.2)

"The construction

else <*unconditional statement*>

is equivalent to

else if true then <*unconditional statement*>"

Again in [12] section 4.6.4.2,

"An element of the form

A **step** *B* **until** *C*,

where *A*,*B*, *C* are arithmetic expressions, gives rise to an execution which may be described most concisely in terms of additional ALGOL statements as follows:

V:=*A*;
L1 : **if** (*V*−*C*) × *sign*(*B*)>0 **then go to** *Element exhausted*;
statement S;
V:=*V* + *B*;
go to *L1*;"

It is assumed that *S* is the statement that has to be repeated, *V* is the control variable and *Element exhausted* refers to the next statement in the program.

Similar definitions are given for other kinds of **for**-statements. But the important point is that certain constructions in the language were used to describe more complicated constructions.

1.4. CONCLUSION

Although certain criticisms have been made of the formal definition of ALGOL 60 it should be remembered that the document defining ALGOL 60 was a fine piece of work. A measure of its success is precisely the amount of study, and therefore criticism, that it has warranted.

One of the effects of the ALGOL 60 Report was that more of the context sensitive information was included in the syntax of programming languages and removed from the semantics. Thus there has been a move towards formality. This has happened in varying degrees as witnessed by the formal definitions of ALGOL W and ALGOL 68.

2. ALGOL W

The programming language ALGOL W was developed around 1966 by N. Wirth and C. A. R. Hoare [18]. It can be regarded as a language which represents

a half-way house between ALGOL 60 and ALGOL 68. Apart from the language itself, its formal definition as given in [2] also represents a half-way house between the formal definition of ALGOL 60 [12] and that of ALGOL 68 [15]. It will therefore be convenient to consider its formal definition and show the developing trends. See also [1, 18].

2.1. SYNTAX

The metalanguage used for formally defining the syntax of ALGOL W is defined by

a set VT of terminal symbols
a set VN of syntactic entities or non-terminals
a set P of syntactic rules or production rules
A syntactic rule has the form

<*a*> ::= *x*

where <*a*> is in the set VN and *x* represents any sequence of terminals and non-terminals. At this stage it is natural to introduce the usual shorthand notation for a set of production rules by making use of the vertical bar. But in ALGOL W the vertical bar is a member of the set VT (it is used for the selection of substrings from strings). Consequently its use would lead to ambiguity unless some care was taken. The solution to this problem was to introduce the syntactic rule

<*bar*> ::= |

and to adopt the convention that whenever | is used as a member of VT it will be replaced by <*bar*>. With this in mind

<*a*> ::= *x* | *y* | . . . | *z*

can be used as an abbreviation for the set of productions

<*a*>::=*x*
<*a*>::=*y*
— — — —
<*a*>::=*z*

Note again the interplay between the design of the metalanguage and the design of the programming language itself. The above rule was introduced to overcome just this difficulty. A similar difficulty is avoided by adding the restriction that

a syntactic entity is denoted by its name (a sequence consisting only of letters, digits and hyphens) enclosed in the brackets < and >.

As a result the rule

<inequality-operator> ::= < | ⩽ | ⩾ | >

is no longer ambiguous.

In many respects the syntactic rules of ALGOL W are like the syntactic rules of ALGOL 60. Thus there are rules such as

<identifier> ::= *<letter>* | *<identifier>* *<letter>*
| *<identifier>* *<digit>* | *<identifier>*

But there is an increased awareness of the idea of type. In ALGOL W it is recognized that in certain constructions it is natural to have only expressions of a particular type. Thus *<integer-expression>* should appear in such places as subscripts, for-statements, when raising to the power, etc. The nonterminal *<real-expression>* also arises naturally as the parameter of **round** or **entier.** The nonterminal *<logical-expression>* appears as the operand of **and**, **not**, **or**, etc. in conditionals and in while-statements.

Thus types play a large part in the syntax and one obtains rules such as

<bound-pair-list> ::= *<bound-pair>* | *<bound-pair-list>*, *<bound-pair>*
<bound-pair> ::= *<lower-bound>* :: *<upper-bound>*
<lower-bound> ::= *<integer-expression>*
<upper-bound> ::= *<integer-expression>*

2.1.1. T-notation

With the introduction of types the number of syntactic rules tends to proliferate. As an example consider

<integer-variable-identifier> ::= *<identifier>*
<real-variable-identifier> ::= *<identifier>*
<logical-variable-identifier> ::= *<identifier>*, etc.

and all the rules are essentially similar. To overcome this problem a convention is introduced. Consider

<T-variable-identifier> ::= *<identifier>*

The *T* stands for type and it is used as follows. If *T* appears in a syntactic rule then it should be understood that this symbol can be replaced by any one of the words

integer real long-real complex long-complex
logical bit string reference

Thus one rule now replaces nine other rules.

When *T* appears more than once in a rule then usually, unless there is a remark to the contrary, the process of replacing *T* must be carried out in a consistent manner, i.e. each occurrence of *T* must be replaced consistently by the same word. To illustrate consider

<simple-T-variable-declaration> ::= *<T-type>* *<identifier-list>*

Replacing *T* by *real* gives

<simple-real-variable-declaration> ::= *<real-type>* *<identifier-list>*

and replacing it by *integer* gives

<simple-integer-variable-declaration> ::= *<integer-type>* *<identifier-list>*

Similarly for other substitutions for *T*. Other parts of the syntax yield the rules

<real-type> ::= **real**
<integer-type> ::= **integer**, etc.

Apart from *T* the symbols *T0*, *T1*, *T2*, . . . also appear in the syntax rules of ALGOL W. The rules regarding *T0*, *T1*, *T2*, . . . are in general slightly different from the rules for *T* and usually only certain combinations of types are allowed. Their use will be illustrated by looking at the syntax of assignment statements and expressions.

2.1.2. Assignment statements

For assignment statements the relevant rules are as follows–

<T0-assignment-statement> ::= *<T0-left-part>* *<T1-expression>* |
<T0-left-part> *<T1-assignment-statement>*
<T-left-part> ::= *<T-variable>* :=

The rules regarding *T*, *T0* and *T1* appear in the semantics and these take the following form (see [2])

"For each left part variable, the type of the expression or assignment variable immediately to the right must be assignment compatible with the type of that variable.

A type *T1* is said to be assignment compatible with a type *T0* if either

(1) the two types are identical (except that if *T0* and *T1* are **string**, the length of the *T0* variable must be greater than or equal to the length of the *T1* expression or assignment), or
(2) *T0* is **real** or **long real**, and *T1* is **integer**, **real** or **long real**, or
(3) *T0* is **complex** or **long complex**, and *T1* is **integer**, **real**, **long real**, **complex** or **long complex**.

In the case of a reference, the reference to be assigned must be null or refer to a record of one of the classes specified by the record class identifiers associated with the reference variable in its declaration."

In summary then the rules regarding *T* are either tabulated in some way as part of the semantics or a consistent substitution rule applies.

2.1.3. Expressions

Expressions are distinguished by means of two quantities, a type and a precedence. As one might expect the type is just the type of the result produced by the expression. The precedence is obtained from the precedence hierarchy imposed on operators in the syntax rules. Table 1 gives the precedence of the various operators.

TABLE 1.

precedence level	operator
1	**or**
2	**and**
3	¬
4	<, < =, =, ¬ =, > =, >, **is**
5	+, −
6	*, /, **div**, **rem**
7	**shl**, **shr**, **
8	**long**, **short**, **abs**

The syntactic entities or nonterminals naming different kinds of expression in the syntactic rules display these two attributes of expressions namely type and precedence. The word "expression" is prefixed by the type and is followed by and integer indicating the precedence level. Hence *<integer-expression-6>* and *<logical-expression-4>*.

In the discussion of expression *T* is replaced consistently in the usual way. But the triplets *T0*, *T1* and *T2* have all to be replaced by the same one of the words

logical
bit
string
reference

or (subject to statements to the contrary) they have to be replaced in accordance with the following *triplet rules*.

(1) Given that *T1* and *T2* can be any combination of the quantities, *integer real* or *complex*, *T0* is given by the table

T1 \ *T2*	*integer*	*real*	*complex*
integer	*integer*	*real*	*complex*
real	*real*	*real*	*complex*
complex	*complex*	*complex*	*complex*

(2) *T0* has the quality *long* if either both *T1* and *T2* have that quality or if one has the quality *long* and the other is *integer*. Apart from *T*, *T0*, *T1* and *T2* it will become necessary to introduce *T3*, *T4*, *T5* – and a corresponding set of rules.

Let us consider now the syntax of expressions in ALGOL W. It is convenient to first consider logical expressions. The idea of associating a precedence with an expression results from the following syntax (denoted for future reference by **).

<logical-expression-1> ::= <logical-expression-1> **or** <logical-expression-2>
<logical-expression-2> ::= <logical-expression-2> **and** <logical-expression-3>
<logical-expression-3> ::= ¬ <logical-expression-4>
<logical-expression-4> ::= <relation>

The syntax defining <relation> makes use of *T6*, *T7*, *T8* and *T9*. These are subject to the following rules:

(A) *T6* and *T7* must either be consistently replaced by any one of the following words

bit, *string*, *reference*

or by any of the words from

complex, *long-complex*, *real*, *long-real*, *integer*.

(B) The symbols *T8* and *T9* must be identically replaced by *string* or must be replaced by any of *real*, *long-real*, or *integer*. The syntax for <relation> now reads

<relation> ::=
<T6-expression-5> *<equality-operator>* *<T7-expression-5>* |
<T8-expression-5> *<inequality-operator>* *<T9-expression-5>* |
<reference-expression-5> **is** *<record-class-identifier>*

and there are the additional rules

<equality-operator> ::= = | ¬=
<inequality-operator> ::= < | < = | > = | >

Consider the syntactic rule for *<logical-expression-1>*. The rule says that this is a sequence of *<logical-expression-2>*'s joined together by **or**. The rule for *<logical-expression-2>* states that such an item is a sequence of *<logical-expression-3>*'s joined together by **and**. Arguing in this manner suggests that an expression such as

a **or** *b*

is not a *<logical-expression-1>* since *b* does not contain **and**. In fact the above is indeed a *<logical-expression-1>* and this follows from the following syntax rules (these are denoted by *** for future reference):

<T-expression> ::= *<T-expression-1>* | *<condition-T-expression>*
<T-expression-1> ::= *<T-expression-2>*
<T-expression-2> ::= *<T-expression-3>*
<T-expression-3> ::= *<T-expression-4>*
<T-expression-4> ::= *<T-expression-5>*
<T-expression-5> ::= *<T-expression-6>*
<T-expression-6> ::= *<T-expression-7>*
<T-expression-7> ::= *<T-expression-8>*
<T-expression-8> ::= *<T-variable>* |
    <T-function-designator> | *<T-constant>* |
    (*<T-expression>*) | *<T-block-expression>*

and finally,

<T-block-expression> ::= *<block-body>* *<T-expression>* **end**

As a result of these rules it follows that an expression which is a sequence of objects joined by **or** is a *<logical-expression-1>*. Similarly a sequence of objects joined by **and** is a *<logical-expression-2>*, etc.

Arguing in the above way it follows that one can associate a meaning with the syntactic entities *<T-expression-i>*. *T* gives the type of the result delivered by

the expression and i gives the precedence of the operator of lowest precedence occurring in the expression and not contained in brackets. Thus

Syntactic Entity	Meaning
T-expression-1	disjunction
T-expression-2	conjunction
T-expression-3	negation
T-expression-4	relation
T-expression-5	sum
T-expression-6	term
T-expression-7	factor
T-expression-8	primary

Note that the syntactic rules denoted by ** assign a precedence to the different operators. Moreover they imply the bracketing – but not the order of evaluation – in expressions such as

a **or** *b* **or** *c* **or** *d*,   *a* **and** *b* **and** *c*,   etc.

The syntactic rules denoted by *** imply that an expression which is valid at one precedence level is valid at all lower precedence levels. Thus an expression valid at precedence level 4 is valid at precedence level 1.

The syntactic definition of arithmetic expressions is similar in many respects to the definition of logical expressions. There are no essentially new ideas involved. However one should note that there are extra rules concerning *T3*, *T4*, *T5*. In fact at this stage much of the semantics is involved in specifying the combination of values that can be taken by the *Ti*'s.

2.2. SEMANTICS

The semantics of ALGOL W are described in a manner similar to those of ALGOL 60. There is of course the added burden of specifying the rules regarding allowable combination of types.

As in ALGOL 60 the effects of certain constructions in the language are described in terms of other usually simpler constructions. This technique is used to a greater extent in ALGOL W. For example

(i) *x*   is equivalent to   **if** *x* **then false else true**
(ii) *x* **and** *y*   is equivalent to   **if** *x* **then** *y* **else false**
(iii) *x* **or** *y*   is equivalent to   **if** *x* **then true else** *y*
(iv) The operator **rem** is defined as follows

$$a \textbf{ rem } b = a - (a \textbf{ div } b) * b$$

Moreover certain standard functions and predeclared variables are defined in a similar though less formal way.

2.3. CONCLUSION

The syntax of ALGOL W is defined in a much more rigorous manner than the syntax of ALGOL 60. The formal mathematical idea of a grammar is apparent in the definition of the metalanguage and also in the semantics.

The syntax still suffers from many of the deficiencies of the formal definition of ALGOL 60. The context dependent requirements are still not included in the syntax. Yet it could be argued that the syntax is the proper place for such requirements since they would normally be checked at compile time.

But other criticisms stem from some of the new innovations. The rules involving *T*, *T0*, etc. are not syntactic rules. But rather they are a new kind of rule from which syntactic rules are derived as a result of substitution. This distinction is not formalized.

The rules regarding the *T*-notation are of course introduced as a matter of convenience rather than necessity. However these rules are in places rather complicated. Much of the semantics contains descriptions of the allowable combinations of types. Again perhaps these should be syntactic rules rather than semantic rules. Note that an increase in the number and variety of types would make these rules even more complicated. However, the consistent substitution rule is rather neat.

Although the *T*-notation has been introduced to avoid redundancy there are still rules such as

<T-expression-1> ::= *<T-expression-2>*
<T-expression-2> ::= *<T-expression-3>*
— — — —
<T-expression-7> ::= *<T-expression-8>*

The same kind of technique should be used here.

Finally note that there is still a certain interplay between the metalanguage and the programming language itself. With careful screening this has been reduced to nuisance value only.

3. ALGOL 68

The programming language ALGOL 68 was first defined in [14], published in 1968. Since that time the language and its method of formal definition have both undergone considerable extensions and revision. The final revised version appeared in 1975 in [15].

In the remainder of this paper we shall consider only the most recent document on ALGOL 68, i.e. [15]. Most of the ideas introduced in the 1968 document are either still present or have been superseded by more recent thoughts.

ALGOL 68 is formally defined by means of four quantities, syntax and semantics together with a standard environment and representations.

In the usual way the syntax provides a method of producing the allowable strings of characters but it does not describe the meaning to be attached to these strings. This last task is performed by the semantics.

If the ALGOL 68 Report had to consist only of a formal definition it would be rather difficult to read. It is useful to have certain explanatory remarks, illustrative examples and so on scattered throughout the Report. These are included by means of, what are called, pragmatic remarks. These are enclosed between { and } and strictly speaking do not form part of the formal definition of ALGOL 68.

Occasional references will be made to the Report. These will take the form of R1.1.3.2a and this just indicates the appropriate section of the Report. (The ALGOL 68 Report itself contains many useful references to other parts of the document. A description of the conventions used is contained in the last paragraph of R1.1.3.4f.)

3.1. REPRESENTATIONS

The sections on representations in the ALGOL 68 Report are concerned with the way in which one writes programs.

To avoid confusion between the metalanguage(s) and the programming language itself no terminals, in the usual sense, appear in the syntax. Indeed this problem is resolved in two stages. As a first step one could arrange that

:=, as used in assignations, appears as the *becomes symbol*;
:=:, as used in identity relations, appears as the *is symbol*;
nil could appear as the *nil symbol* and so on.

In the syntax rules in the Report, however, one finds not the *becomes symbol* or the *is symbol* but rather the *becomes token*, the *is token*, etc. A *token* is merely a symbol preceded, if required, by comments or pragmatic remarks (as they appear in programs as opposed to the formal definition). Thus each of

co *this is a comment* **co** :=
:=:
and
pr *switch on overflow check* **pr** **begin**

is an example of a token (assuming that the text enclosed by **pr** constitutes an acceptable pragmatic remark).

3.2. THE STANDARD ENVIRONMENT

In writing in many programming languages the user usually assumes the existence of a set of standard functions such as *sin*, *cos*, *read*, *print*, etc. which

he can use. The compiler is assumed to know all about these and incorporate the appropriate code.

In ALGOL 68 the story is similar. But the set of standard objects known to the compiler is increased since this now includes modes, operators, constants and variables as well as procedures. The standard environment contains a description of all such items and a complete chapter of the ALGOL 68 Report, Chapter 10, is devoted to this.

In the syntax one will not find terminals such as + or even *plus token*. Instead one finds something almost as vague as *operator*. This could then be replaced by either a standard operator (as defined in the standard environment) or a user defined operator (as declared by a programmer).

The method used to describe the standard environment is reminiscent of the method used to describe the standard environment of ALGOL W. Thus the various items are defined either in terms of ALGOL 68 itself or, wherever appropriate, by appealing to informal comments. The entire section on transput (i.e. input/output) is contained in the standard environment and all the various transput procedures including formatted, unformatted and binary transput are defined in terms of ALGOL 68 itself.

The standard environment is itself divided into various preludes, postludes and tasks. The reason for the divisions can be illustrated by considering preludes. The standard prelude contains most of the usual declarations of constants, procedures, operators, etc. All these are common to all ALGOL 68 programs. Indeed only one copy of these need exist in a computer though there may be several different ALGOL 68 programs being executed at a given time. There is no danger of one program interfering with another. On the other hand each program has its own particular prelude and this contains declarations of all objects accessible only by that program. A particular prelude would include, for example, the declarations of the standard transput files.

3.3. THE SYNTAX OF ALGOL 68

The method of describing the syntax is more complicated than the previous methods. The reason for this stems from the fact that many of the context-sensitive details are now present. The syntax should now be seen as containing all the restrictions, etc. which a compiler can attempt to check. It contains, for example, mode or type information together with information about coercions (automatic mode changes). It describes also the process of associating occurrences of identifiers, etc. with their corresponding declaration.

But rather than look at the complete syntax immediately this task will be broken down into several easier stages:

(i) The syntax with all the mode information including coercions and the information relating applied and defining occurrences of identifiers will be removed.

(ii) To the syntax obtained in (i) above the mode information will be added.

This will also necessitate a look at the way in which coercions appear in the syntax.

(iii) The process of relating applied and defining occurrences will be added.

(iv) The complete syntax will be considered.

This then describes in rough detail how the syntax will be introduced.

3.3.1. Syntax with modes, context conditions, predicates, etc. removed

Consider the (modified) syntax rule

UNIT :: *assignation*; *identity relation*; *routine text*;
jump; *skip*; TERTIARY.

(In this section the difference between upper case and lower case letters will be, for the most part, ignored.)

This syntax or syntactic rule states that a UNIT (i.e. a unitary clause) is defined to be either an assignation, or an identity relation, or a routine text or a jump or a skip or, finally, one of the set of constructions included under the heading of a TERTIARY. Thus one can regard

:: as a shorthand for "is defined to be"

; as a shorthand for "or"

and . denotes the end of the rule.

Other symbols which will be used in this way are

: which also is a shorthand for "is defined to be"

and , which is a shorthand for "followed by"

The reason for having both : and :: will be discussed at a later stage. However to illustrate the use of the comma and the colon consider the syntax rule for an assignation

assignation : *destination*, *becomes token*, *source*.

Thus an *assignation* is defined to be a *destination* followed by a *becomes token* followed by a *source*.

Note that in the more familiar BNF notation as used in the formal definition of, say, ALGOL 60 these two rules would have appeared as something like

<unit> ::= *<assignation>* | *<identity relation>* | *<routine text>* |
<jump> | *<skip>* | *<tertiary>*
<assignation> ::= *<destination>* := *<source>*

and remarks about commentary and/or pragmatic remarks would have appeared in the semantics rather than in the syntax.

An *assignation* is therefore similar to the ALGOL 60 concept of an assignment statement though it is more general since a value is associated with each assignation and this in turn can be used in expressions.

To discuss some of the concepts introduced by the syntax rules it will be assumed that the reader understands at least vaguely the form of the ALGOL 68 constructs, formula, generator, slice, call, selection and identity relation.

Example illustrating formulae, generators, etc.

(i) Formula is the ALGOL 68 word for expression;

$$a+b\times c,\ -d,\ (a:=b)\times c+b$$

are all formulae

(ii) Generators are used when space has to be created. A generator takes the form of the symbol **loc** (indicating local space) or the symbol **heap** (indicating global space) followed by a declarer;

loc real , **loc** [4] **int**

are generators.

(iii) Slices are used in subscripting or in selecting subarrays from larger arrays. Thus

$$x[i]\ ,\quad y[4:10]$$

are slices.

(iv) Calls are used to invoke routines:

sin(*x*), *print* (*x*↑2 + 4)

are calls.

(v) Selections are used when it is required to access one particular field of a structure. *age* **of** *man* would constitute an example of a selection.

(vi) ALGOL 68 allows the existence of variables whose values are other variables. Identity relations are used to compare the values of such variables for equality. Thus

x **is** *y* *x* **isnt** *y*

are both examples of identity relations

Consider now a larger set of the syntax rules. These include not only the rule for UNIT and the rule for an assignation but also the rules for destination, source, TERTIARY, etc. (The corresponding unmodified rules are in R5.1 and R5.2). To avoid complexities the rules for routine texts, jumps, skips, nihils, casts and format texts are all ignored for the present.

syntax of UNIT

UNIT:: *assignation*; *identity relation*; *routine text*;
 jump; *skip*; TERTIARY.

syntax of TERTIARY

TERTIARY:: *formula*; *nihil*; SECONDARY.

syntax of SECONDARY

SECONDARY:: *generator*; *selection*; PRIMARY.

syntax of PRIMARY

PRIMARY:: *slice*; *call*; *cast*; *denoter*; *format text*;
identifier; ENCLOSED clause.

syntax of assignation

assignation: *destination*, *becomes token*, *source*.
destination: TERTIARY.
source: *unit*.

syntax of identity relation

identity relation: TERTIARY1, *identity relator*, TERTIARY2.
identity relator: *is token* ; *is not token*.

As one might expect the appearance of TERTIARY1 and TERTIARY2 implies that different TERTIARY's can appear on the different sides of an identity relator.

For completeness the (modified) syntax of slices, selections and calls are given below:–

slice : PRIMARY, *indexer bracket*.

(the indexer bracket is just the square brackets together with what is enclosed therein)

selection: *identifier*, *of token*, SECONDARY.
call: PRIMARY, *parameter pack*.

(the parameter pack is just the parameters separated by commas together with the round brackets which surround them).

Example on UNIT, PRIMARY, etc.

(a) Each of the following constructions is a PRIMARY
  (i) $x[i]$ is a slice and therefore a PRIMARY
  (ii) $sin(pi/2)$ is a call and so a PRIMARY.
(b) Each of the following constructions is a SECONDARY
  (i) *age* **of** *boy* is a selection and so a SECONDARY.
  (ii) **loc real** is a generator and therefore a SECONDARY.
  (iii) $sin(pi/3)$ is a call and thus a PRIMARY and therefore a SECONDARY.
(c) Each of the following constructions is a TERTIARY
  (i) $a \times b+c$ is a formula and so a TERTIARY
  (ii) $sin(pi/3)$ is a PRIMARY and therefore a SECONDARY and therefore a TERTIARY.
(d) The following are UNITS.
  (i) $x := y+a \times b$ since it is an assignation
  (ii) $sin(pi/3)$ is a call and a UNIT
  (iii) $x :=: y$ is an identity relation and so a UNIT
  (iv) $a+b \times c$ is a TERTIARY and therefore a UNIT.

These examples and this introduction to the syntax of ALGOL 68 allow the implied bracketing to be inserted in certain constructions. Consider the assignation

$$x := y := z$$

Should this be interpreted as $(x := y) := z$ or as $x := (y := z)$ or indeed, in some other way? Consider the first interpretation and consider its implications. This implied bracketing would have the consequence that an assignation could appear on the left of the *becomes symbol* (or the *becomes token*) in an assignation. But from the syntax rule for an assignation, at most a TERTIARY – a destination is defined to be a TERTIARY – can appear on the left of the *becomes symbol*. And an assignation is certainly not a TERTIARY. So this implied bracketing would not take place.

The interpretation $x := (y := z)$ is satisfactory. For a source can be any UNIT and from the syntactic rule for a UNIT an assignation is certainly a UNIT. In fact this is the interpretation that would be placed on this sequence of symbols. ALGOL 68 is designed in such a way that each construction can be so analysed in at most one way. Note that no mention has been made of mode. In ALGOL 68 this parsing is actually independent of the modes of the constituent objects.

Some further examples of the mode independent parsing of ALGOL 68 are given in the next example.

Example on mode independent parsing

(i) $a + b[i]$
is parsed as $a+(b[i])$ and not as $(a+b)[i]$. The reason for this lies in the fact that one can subscript a PRIMARY. b is a PRIMARY but $a+b$ is a TERTIARY.

(ii) c of $d[i]$
is parsed as c of $(d[i])$ and not as $(c$ of $d)[i]$. Again since only a PRIMARY can be subscripted the first possibility is certainly legal. For $d[i]$ is a slice and therefore a PRIMARY. But the interpretation $(c$ of $d)[i]$ is not legal since c of d being a selection is thereby a SECONDARY.

(iii) $a := b :=: c$
is parsed as $a := (b :=: c)$ and not as $(a := b) :=: c$. The reasoning is similar to that employed in parts (i) and (ii) above.

(iv) $a :=: b := c$
is syntactically illegal since it is not possible to deduce from the syntax rules any legal interpretation.

(v) *day* of *month* of *year*
is parsed as *day* of (*month* of *year*).

This then completes a first glimpse of the syntax of ALGOL 68.

Note that the parsing indicated by means of the above modified syntax rules is completely mode independent. For it has been possible to deduce the implied bracketing regardless of the modes of the objects being manipulated.

However a comprehensive study of the syntax involves, apart from other concepts, considerations of the modes and the changes of mode, i.e. coercions, that occur in ALGOL 68 programs. The next stage of the investigation of the formal definition of ALGOL 68 introduces this.

3.3.2. The meaning of capital and small letters

As a prologue to the introduction of modes, etc. into the syntax rules discussed in section 3.3.1 above the distinction between capital letters and small letters and between the symbols :: and : will be clarified. That these are different was ignored in the previous section.

Strictly speaking the constructions of the programming language ALGOL 68 are defined by means of certain kinds of syntax rules which hereafter will be called production rules of the language. These production rules contain no capital letters nor do they contain the symbol :: used to mean "is defined to be". An example of such a rule is (taken from the syntax of identity relations)

identity relator : *is token* ; *is not token.*

Note that there is no interplay possible between the method of formal definition and programming language since ; and : would appear in the syntax as the *go on symbol* and the *label symbol* (*colon symbol* , *routine symbol* or *up to symbol*) respectively.

Example production rules of the language

The examples listed below are taken from the part of the Report dealing with denotations, i.e. Chapter 8.

(i) *boolean denotation* : *true token* ; *false token.*
This merely states that, ignoring comments or pragmatic remarks, one writes a boolean as the *true symbol* or as the *false symbol.*

(ii) *void denotation* : *empty token.*

(iii) *integral denotation* : *fixed point numeral.*
fixed point numeral : *digit cypher sequence.*

Numerous other examples of the production rules of the language are contained in Chapter 8. In several cases *symbol* appears rather than *token*. This merely implies that comments and pragmatic remarks cannot appear in these positions.

3.3.2.1. Definitions involving small letters. It is convenient at this stage to define some terms which are used frequently and which allow one to talk more freely about rules involving small and/or capital letters.

A *protonotion* is a non-empty sequence of small letters together with (and). These can be punctuated by spaces, etc. if required. The spaces have no significance. The significance of the characters (and) will become apparent later. Further mention of them will be ignored until then.

Example protonotions

Each of the following is an example of a protonotion

(i) *identity relator* (ii) *variable point numeral*

(iii) *variable point* (iv) *empty cup*

Examples (i) and (ii) above are more interesting in that they appear on the left hand side of production rules (for identity relations and floating-point numerals). For this reason it is convenient to highlight them by making the following definition:

a *notion* is a protonotion for which there is a production rule of the language. (In this definition of notion a rule means a rule which is part of the syntax of the ALGOL 68 Report, not a modified rule.)

Example notions

The following examples of notions are again taken from the section on denotations.

(i) *variable point numeral*
(ii) *stagnant part*
(iii) *plusminus*

One other group of protonotions which deserve special mention is the set of symbols. The strict definition of a symbol is:

a *symbol* is a protonotion ending in *symbol*.

Examples of symbols are readily available: *true symbol*, *is symbol*, *point symbol*, etc. The symbols are just the terminals of the grammar.

Thus the interesting protonotions are just the notions and the symbols or tokens.

The format of the production rules of the language has now been given. One can describe the right hand side of such a rule by saying that it consists of a sequence of *alternatives*, the alternatives being separated by means of semi-colons.

Example alternatives

Consider the rules taken from the syntax describing a floating point numeral

exponent part . times ten to the power choice,power of ten.
times ten to the power choice : times ten to the power symbol ; letter e symbol.

There is only one alternative in the first of these rules i.e.

times ten to the power choice, power of ten.

In the second rule there are two alternatives namely

times ten to the power symbol and *letter e symbol.*

Each alternative can now be described by saying that it consists of a sequence of *members* separated by commas.

Example members
Using the production rules given in the previous example it follows that each of

times ten to the power choice and *power of ten*

are members as are

times ten to the power symbol and *letter e symbol.*

3.3.2.2. Definitions involving capital letters. As mentioned above all the syntax rules of the language itself are characterized by the fact that no capital letters are used. But in the syntax parts of the Report there are rules such as

plain denotation : PLAIN *denotation* ; *void denotation.*

Now PLAIN is written in a different type font and consequently this rule is not a rule of the strict language. However rules such as this are used to produce the production rules of the language. In fact just as the production rules of the language produce programs so rules such as the above are used to produce the production rules of the language.

In the formal definition of ALGOL 68 there is another set of rules giving a definition of the words in capital letters. These rules form the grammar for another language called the *metalanguage*. Note that this is now a different use of the term "metalanguage" from that used earlier in discussing ALGOL 60 and ALGOL W. Here the metalanguage is used for producing the production rules of the language ; previously the production rules themselves constituted the metalanguage.

For this metalanguage there are analogues of production rules, symbols, members, etc. Moreover these rules infer an implied bracketing just as the earlier production rules inferred the implied bracketing described previously.

A typical rule of the metalanguage is

PLAIN::INTREAL;*boolean* ;*character.*

i.e. PLAIN is defined to be INTREAL or one of the protonotions *boolean* or *character*. Another rule is

INTREAL :: SIZETY *integral* ; SIZETY *real.*

These rules of the metalanguage are called *metaproduction rules*. Putting together the two rules above it follows that a SIZETY *real* is a derivation of the metanotion PLAIN. Note that in these metaproduction rules the double colon appears for "is defined as". In fact it is the presence of the double colon that characterizes metaproduction rules.

Corresponding to the earlier idea of a notion is the idea of a *metanotion.* Thus a metanotion is a sequence of capital letters for which there is a

metaproduction rule. And corresponding to the earlier idea of a symbol is the idea of a protonotion. These protonotions are of a special kind and are therefore called *terminal metaproductions*.

Example metanotions

(i) PLAIN and INTREAL are metanotions since they appear on the left of the double colon in the above metaproduction rules.

(ii) Other metanotions are

(a) MOID (b) THING
(c) SIZE

Example terminal metaproductions

(i) *boolean* and *character* are terminal metaproductions of the metanotion PLAIN.

(ii) *long long integral* is a terminal metaproduction of INTREAL. For consider the metaproduction rules

INTREAL :: SIZETY *integral* ; SIZETY *real*.
SIZETY :: *long* LONGSETY ; *short* SHORTSETY ; EMPTY.
LONGSETY :: *long* LONGSETY ; EMPTY.

The ALGOL W analogue of this metalanguage is just the set of rules which were contained in the semantics of ALGOL W and gave the allowable combinations of types that could be associated with *T*, *T0*, *T1*, etc. So this aspect of the semantics of ALGOL W has been transferred to the syntax of the metalanguage. Note that in keeping with this observation the metanotions introduced above all involved types or modes of some kind. However there are also other situations in which the metalanguage is used. It is used for example to overcome the problem of having a set of syntax rules such as the ALGOL W rules

<T-expression-1>::=<T-expression-2>
<T-expression-2>::=<T-expression-3>, etc.

This aspect of the metalanguage is discussed more fully in section 3.3.6.2 of this paper.

3.3.2.3. Consistent substitution. It was mentioned earlier that the metalanguage is used to produce the syntax rules of ALGOL 68. To explain this consider again

plain denotation : PLAIN *denotation* ; *void denotation*.

By using the metaproduction rule for PLAIN the above rule can be used to generate several production rules. In this rule for *plain denotation* PLAIN can be replaced by any of its terminal metaproductions. This replacement then gives a

new set of production rules. For instance replacing PLAIN by the terminal metaproduction *character* yields

plain denotation : *character denotation*; *void denotation*.

Replacing PLAIN by *long long integral*, another terminal metaproduction of INTREAL, gives

plain denotation : *long long integral denotation* ; *void denotation*.

In this way the different grammars can be combined to yield production rules. If several different metanotions exist within the one rule then each metanotion should be replaced by one of its terminal metaproductions in order to yield a production rule. If a particular metanotion appears more than once in a rule this metanotion must be replaced *at each of its occurrences in the rule* by the same terminal metaproduction. This is termed consistent substitution and is reminiscent of a corresponding rule in ALGOL W.

To illustrate these ideas consider

SIZE INTREAL *denotation* : SIZE *symbol*, INTREAL *denotation*.

This discussion will make use of the earlier metaproduction rule for INTREAL together with

SIZE :: *long* ; *short*.

Applying consistent substitution SIZE has to be replaced throughout by either *long* or by *short*. It would be illegal to replace SIZE by *long* in one case and *short* in the other. If *long* is chosen this results in

long INTREAL *denotation* : *long symbol*, INTREAL *denotation*.

Similarly replacing INTREAL by one of its terminal metaproductions, say *long integer*, gives

long long integer denotation : *long symbol*, *long integer denotation*.

There are cases where consistent substitution is not necessary and it is fairly clear that it is not necessary. Consider the modified rule for an identity relation

identity relation : TERTIARY1, *identity relator*, TERTIARY2.

The appearance of the 1 and the 2 indicates that consistent substitution is not necessary.

3.3.2.4. Hyper-rules. The syntax used in defining ALGOL 68 then consists basically of a two level grammar – often called a production van Wijngaarden grammar. At one level are the rules of the programming language. At a different level are the metaproduction rules.

But there is also a third kind of rule that is freely used. Consider

SIZE INTREAL *denotation*: SIZE *symbol*, INTREAL *denotation*.

This is neither of the two kinds of rules specified above. Yet it is the type of rule that is used to produce the production rules by using the metalanguage. Such a rule is called a *hyper-rule*. The terms hyperalternative and hypernotion all have their expected meanings.

Example hyperrules

(i) SIZE INTREAL *denotation* is a hypernotion (as witnessed by the above rule).
SIZE *symbol*, INTREAL *denotation* is a hyperalternative

(ii) *plain denotation* : PLAIN *denotation* ; *void denotation*. is a hyperrule. Consequently *plain denotation* is a hypernotion and both PLAIN *denotation* and *void denotation* are hyperalternatives.

The term hypernotion could be more precisely defined as follows. A hypernotion is a possibly empty sequence each of whose elements is either a small letter or a metanotion.

In order to become familiar with the Report it is important to have a thorough grasp of the meanings of the different metanotions. It is necessary to do this in order to be able to read the Report with relative ease.

To enable the reader in this task Appendix A attempts to explain the meaning of all the metanotions in simple terms. However the reader should attempt to deduce some of these for himself.

Example on the use of the metanotions and hypernotions

ALPHA is defined to be any small letter.

NOTION is defined to be any sequence of (one or more) small letters. Consider the rule

NOTION *option* : NOTION ; EMPTY.

Replacing NOTION by its terminal metaproduction *plusminus* gives

plusminus option : *plusminus* ; EMPTY.

This rule is required in the syntax for the exponent parts of real numerals.

Similarly replacing NOTION by its terminal metaproduction *integral part* gives

integral part option : *integral part* ; EMPTY.

(This rule is also required in the syntax of real numerals.) Similarly uses of *option* are required in other rules.

From the rule

NOTION *sequence* : NOTION; NOTION, NOTION *sequence*.

one can deduce that a sequence is a list of (one or more) objects.

Having introduced these ideas the reader should now be able to read several sections of the syntax. For instance he should be able to read and understand almost all of the syntax contained in Chapter 8 of the Report.

3.3.2.5. Paranotions. Now at certain stages of the syntax the syntactic rules become fairly complex and some of the hyper-notions become rather lengthy. For this reason certain abbreviations in hyper-notions are allowed and these abbreviations are called paranotions (paraphrased hypernotions?). To be rigorous certain rules are laid down in the Report and these state just what abbreviations are allowed. But this is not discussed in depth here. Roughly speaking instead of having to talk about real expressions, integer expressions, etc. one is allowed to talk about expressions. "expressions" then becomes a paranotion. In fact the paranotions provide the vocabulary which one would normally use in discussing aspects of ALGOL 68.

Example paranotions
To give some idea of what constitutes a paranotion some examples are given below.

(i) *formula*	(ii) *constant*	(iii) *declaration*
(iv) *lower-bound*	(v) *operand*	(vi) *in*-CHOICE-*clause*

Some of these paranotions actually have syntax rules. If an asterisk is placed beside the rule in the Report it implies that this is included merely as an aid to the reader, not as a rule required at some other part of the definition of the syntax of the language.

Summing up then the syntax of ALGOL 68 is defined by means of a two level grammar. The hyper-rules produce the production rules of appealing to the metaproduction rules. The fact that these are two levels of grammar allows one to make certain comparisons. Just as the production rules for ALGOL 68 generate an infinite number of possible programs so the metaproduction rules generate an infinite number of production rules. For instance INTREAL has an infinite number of terminal metaproductions. Moreover one can draw a table with the corresponding ideas indicated.

small letters	capital letters
notion	metanotion
production rule	metaproduction rule
symbol	protonotion, in particular, terminal metaproduction
alternatives	hypernotions
	::

One can now attempt to take the analogy a little further and seek two levels of implied bracketing.

The sections of the Report dealing with syntax generally fall into two classes. The first class gives the metaproduction rules labelled with capital letters. These are to be used with the hyperrules that follow in the second class. The hyperrules are labelled with small letters.

3.3.3. Inserting modes into the syntax

Until now the syntax that has been considered has been of a relatively simple nature. In general the syntax used in the ALGOL 68 Report is fairly complex. The intention here is to start from what has already been discussed and gradually build up to the full complexity of the complete syntax. In this section the introduction of modes into the syntax is considered. This in turn will lead to the introduction of several other related concepts.

3.3.3.1. Modes associated with constructs. In learning ALGOL 68 one becomes accustomed to associating a value and a mode with each unitary clause. The mode is just the mode of the result produced by that construction. This same mode appears in the syntax. To illustrate consider the identity relation. This always produces a boolean result. Consequently the syntax of identity relations can now be expanded from the earlier simplified syntax rule

> *identity relation* : TERTIARY1, *identity relator*, TERTIARY2.

to the more complete (though still simplified) rule

> *boolean identity relation:reference to* MODE TERTIARY1, *identity relator, reference to* MODE TERTIARY2.

Note the presence of *reference to* before MODE. Since a mode has now to be associated with each construction a mode has to be associated with TERTIARY1 and the TERTIARY2. The rule about consistent substitution implies that, since *reference to* MODE appears before TERTIARY1 and also before TERTIARY2, both sides of an identity relator must yield objects of the same mode and these objects must be variables.

Thus the syntax rule provides much more information. As far as mode information is concerned it indicates that

an *identity relation* always produces a result of mode **bool**

and

each side of the *identity relator* must produce a variable of the same mode.

As another example of the introduction of modes consider the again simplified hyper-rule defining a generator

reference to MODE LEAP*generator* : LEAP *token*, *actual* MODE *declarer*.

and the hyper-rule defining a sample generator

reference to MODINE LEAP *sample generator* : LEAP *token*, *actual* MODINE *declarer*.

Note that consistent substitution is required for the metanotions MODE and LEAP in the first of those rules and for MODINE and for LEAP in the second.

These rules again indicate that the value associated with a generator is a variable. In the first of these two rules this is indicated by the presence of *reference to* MODE and in the second case by the presence of *reference to* MODINE.

Now consider a third example to illustrate the presence of modes. This next example relates to assignations and this will introduce some new ideas. Note that if x is of mode **ref real** and if n is of mode **ref int** the following are all perfectly legal assignations

(a) $x := 1.4$
(b) $x := 1$
(c) $x := n$

In cases (b) and (c) certain coercions are necessary in order to fulfill the condition, obtained from the syntax rule below, that if x is of mode **ref real** the right hand side must yield a result of mode **real**.

The (still modified) rule for an assignation reads

REF *to* MODE *assignation* : REF *to* MODE *destination*, *becomes token*, MODE *source*.

As usual consistent substitution is required and
the value associated with an assignation has mode REF *to* MODE
the destination produces a result of mode REF *to* MODE
the source gives a result of mode MODE.

Modified hyperrules for destination and source
read

REF *to* MODE *destination* : REF *to* MODE TERTIARY.
MODE1 *source* : MODE2 *unit.*

This last rule for a source covers the possibilities given above in the assignation involving the destination of mode **ref real**. For from the rule it follows that since consistent substitution is not necessary due to the presence of the 1 and 2 one has the rules

real source : *real unit.*
real source : *integral unit.*
real source : *reference to integral unit.*

Consequently the above assignations are all covered. However in this simplified form the rule also implies that an assignation such as x := "A" would be legal. It is not. So further information must be incorporated somewhere. This particular problem will be left for the moment but its solution is related to the next stage of making the syntax more sophisticated. The next step is to introduce the allowable coercions, i.e. automatic mode changes. It can then be shown that the above assignation is illegal for suitable coercions just do not exist.

3.3.3.2. Strength of syntactic position. The first step in the process of introducing the allowable coercions into the syntax is to introduce the strength of the syntactic position. It is perfectly natural to do this since the strength of the position plays some part in determining just what coercions are allowed.

The rules for sources and destinations can be made to incorporate this information as follows

REF *to* MODE *destination* : *soft* REF *to* MODE TERTIARY.
MODE1 *source* : *strong* MODE2 *unit.*

The presence of *soft* in the first rule indicates that a destination is a soft syntactic position. Similarly the presence of *strong* indicates that a source occupies a strong syntactic position. Again the rule for subscripts of slices when simplified yields

subscript : *meek integral unit.*

implying that a subscript occupies a *meek* syntactic position.

In more complicated cases one can also note from the rules the presence of the strength of the syntactic position. For instance in

the rule for a slice the PRIMARY (i.e. x in $x[i]$) occupies a weak position;

the rule for a selection the SECONDARY (i.e. *man* in *age* **of** *man*) occupies a weak position;

and

the rule for an operand, an operand (e.g. *a* in *a+b*) occupies a firm position.

By combining the strength of the syntactic position and the mode information in the rules it is possible to provide via the syntax information about the coercions. (This is contained in Chapter 6 of the Report).

3.3.3.3. MORF and COMORF. In order to understand Chapter 6 of the Report one must be aware of certain points concerning coercions. The first such point is the reason behind the fact that the set of constructions included within the metanotion FORM (see Appendix A, number 32) is divided into two distinct classes represented by the metanotions MORF and COMORF. The trouble arises from the two coercions deproceduring the voiding. In applying coercions to reduce an object to mode **void** there would, unless some precautions were taken, be some ambiguity about whether a construction yielding an object of mode **proc void** had to be deprocedured or voided. In fact **proc void** is not the only such mode but this is the only one that will be considered for the moment.

The difference between the two coercions lies in the fact that whenever deproceduring occurs it implies an activation of the procedure involved. Voiding implies that the routine is not activated. Consequently it is important that one should be clear about whether deproceduring or voiding takes place. The reason for the classification of FORMs into MORFs and COMORFs is just this distinction. More precisely the results produced by COMORFs are voided without deproceduring and therefore without activation of any procedures contained therein. MORFs are voided by deproceduring, whenever deproceduring is appropriate. It is important to realise that this trouble of having to distinguish between MORFs and COMORFs arises only when the mode has to be reduced to **void** and only when procedures are involved in particular ways. A fuller explanation of the meaning of this last remark is given later.

The (simplified) metarules for MORFs and COMORFs are

MORF :: *selection* ; *slice* ; *routine text* ; *formula* ; *call* ;
applied identifier.
COMORF :: *assignation* ; *identity relation* ; *generator* ; *cast* ;
denoter ; *format text*.

Example COMORFs

Let *p* be of mode **ref proc void** and *s* be of mode **proc void**.
Consider

.... ; *p* := *s* ; ....

Due to the presence of the semi-colon this assignation delivers a result which has to be voided. The result is *p* of mode **ref proc void**. The mode is reduced to **void** by means of the coercion voiding since an assignation is a COMORF. Thus this assignation takes place without the activation of any procedure.

Example on MORFs

Suppose that *dump* is a procedure with mode **proc void**. The effect of *dump* might be to print out the value of a set of variables. Suppose that

. . . . ; *dump* ; . . . .

appeared in a program. Then the object of mode **proc void** has to be reduced to mode **void**. This is done by deproceduring and hence the procedure *dump* is activated. The reason that deproceduring takes place is that the above is an applied occurrence of the identifier *dump* and is therefore a MORF.

Note that the distinction is a perfectly natural subdivision. If any of the constructions included under MORF produced a result of mode **proc void** and if this had to be voided (as *dump* above) then it would be perfectly natural to expect activation of the procedure. In the case of COMORFs it would be natural not to expect activation. Note that the COMORFs include identity relations which can never produce a result of a mode such as **proc void**.

There are several points that should be made in order to clarify the above remarks and to tie up some loose ends. The distinction between MORFs and COMORFs is not always appropriate. In fact it is appropriate only if procedures are involved. If a MORF happens to deliver a result which does not begin with **proc** then there is no need to worry about deproceduring. But there is another point to be considered.

The earlier discussion was conducted in terms of MORFs delivering results of mode **proc void**. But **proc void** results are not the only modes that are relevant. If a MORF after possible deproceduring and dereferencing produces a result whose mode is included in the metanotion NONPROC then it can be voided. NONPROC includes all modes except **proc moid, ref proc moid**, **ref ref proc moid**, etc. (**moid** includes all modes, even **void**). Thus none of the modes

ref proc real, ref ref proc void or **proc real**

is included in the set NONPROC.

So all NONPROC MORFs can be voided. But MORFs of other modes can never be voided.

These latter remarks have been included so that they would provide an introduction to the discussion about including within the syntax the various coercions that can take place in different syntactic positions. This task can now be considered in some detail.

3.3.3.4. Coercions. Consider the assignation

$$x := i$$

where i is of mode **ref int** and x is of mode **ref real**. Then the coercions which take place here are, in the usual way, dereferencing followed by widening. Thus i

is dereferenced and the result widened. Consider this in more detail. Essentially there are two different objects associated with the right hand side of this assignation. There is

(i) the reference to integral identifier *i*, i.e. the object before coercions take place

(ii) the real which is produced as a result of the coercions.

The object at stage (i) is called a coercend since it is the object to which coercions are applied. The object at stage (ii) is called a coercee being the final object.

To follow the syntax of coercions in Chapter 6 of the Report write down at each stage of the coercion process in the above example

(i) the last coercion applied

(ii) the result produced by this last coercion

(iii) the type of construct producing the result being coerced.

Of course item (iii) will remain the same throughout.

Consider again the above example. Initially the coercend *i* is a

reference-to-integral-applied-identifier.

The first coercion to act is dereferencing. Writing down the three items listed above then gives

dereferenced-to-integral-applied-identifier.

Thus dereferencing is the coercion applied and dereferencing produces an integral result. The second coercion is widening and writing the relevant three items results in

widened-to-real-applied-identifier

In summary then we have at the various stages

(a) *reference-to-integral-applied-identifier* (initial situation)

(b) *dereferenced-to-integral-applied-identifier*

(c) *widened-to-real-applied-identifier*

(d) *strong-real-applied-identifier-coercee* (final situation)

Stage (d) merely replaces the final coercion by the strength of the syntactic position and adds *coercee*.

Thus the coercion process works successively from stage (a) through to stage (d). Stage (a) produces the coercend and stage (d) produces the coercee.

The syntax describing this coercion process works in the opposite direction, i.e. it starts at stage (d) and produces stage (c) which in turn produces (b) and this eventually produces (a). To illustrate note that an applied identifier is essentially a FORM (and also a MORF). Using the rule

strong MOID FORM *coercee*: STRONG MOID FORM.

and replacing STRONG by *widened to* (see Appendix A, number 116), as is allowed by the substitution rules already considered, gives the step from stage (d) to stage (c). Here MOID is replaced by *real*.

To reach stage (b) from stage (c) consider the rule

widened to SIZETY *real* FORM: MEEK SIZETY *integral* FORM.

with SIZETY replaced by EMPTY. It then reads

widened to real FORM: MEEK *integral* FORM.

Replacing MEEK by *dereferenced to* (see Appendix A, number 52) gives stage (b), i.e.

widened to real FORM: *dereferenced to integral* FORM.

To reach stage (a) from stage (b) consider the rule

dereferenced to MODE1 FORM: MEEK REF *to* MODE2 FORM.

with MODE1 and MODE2 replaced by *integral* and REF by *reference*. This gives

dereferenced to integral FORM: MEEK *reference to integral* FORM.

Finally replace MEEK by *unchanged from* and use the rule

unchanged from reference to integral FORM: *reference to integral* FORM.

This then completes the required chain of coercions.

The above is just one simple illustration of the syntax rules for coercions. Other examples will further illustrate them. These are left to the reader. This task may be helped by what now follows.

3.3.3.5. Coercion chart. A careful study of the coercion rules allows one to produce a chart indicating the allowable coercions. This is given in Fig. 1.

One starts at the top of the chart with the coercend and one must emerge before the appropriate box, indicating the strength of the syntactic position, with the appropriate coercee. Thus in a meek position the appropriate box is marked MEEK, in a strong position the box is marked STRONG, etc. The route adopted then indicates the coercions that have to be applied. Some notes on the chart are given after the chart.

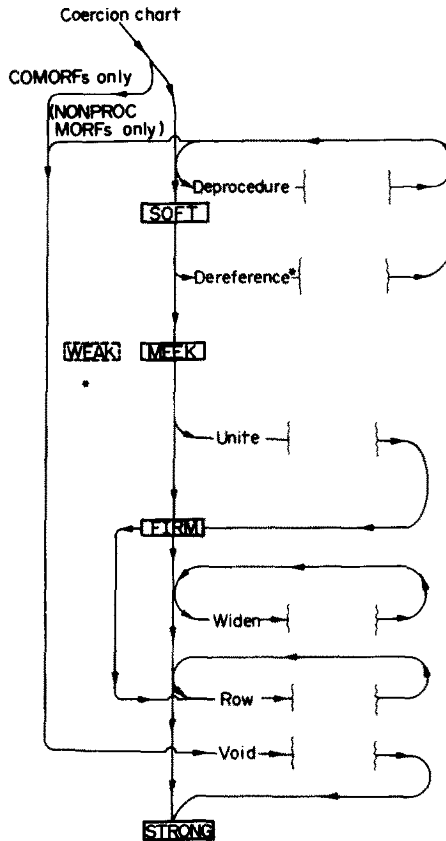


Fig. 1

Notes on the coercion chart

(a) In, for instance, a soft syntactic position one can never go below the box marked SOFT. Similarly for the other syntactic positions. Thus dereferencing is not allowed in a soft position.

(b) Weak positions are essentially the same as meek positions but only weak dereferencing can be applied, i.e. the last **ref** cannot be removed. Hence the * beside WEAK and dereferencing.

(c) The note, NONPROC MORFs only, at the exit from deproceduring (and after possible dereferencing) indicates that one can take this route only if the construction producing the result whose mode is being coerced is included in the set of MORFs (the mode so far produced must be included in NONPROC).

(d) Note one cannot apply a strong coercion before uniting. A list of the allowable coercions appears in Fig. 2.

widening (see note 1 below)	s int to s real s real to s compl s bits to [ ] bool s bytes to [ ] char
deproceduring	proc moid to moid
dereferencing	REF moid to moid
voiding	NONPROC MORF to void mode COMORF to void
uniting	mode to union (. . . . , mode, ...)
rowing (see note 2)	reflexety mode to refety [ ] mode reflexety [ ] mode to refety [,] mode reflexety [,] mode to refety [,] mode etc.

Note 1. s denotes size i.e. any sequence of longs or any sequence of shorts.

2. reflexety and the corresponding refety are given in the table below

reflexety	*refety*
EMPTY	EMPTY
reference to	*reference to*
transient reference to	*transient reference to*
reference to flexible	*transient reference to*
transient reference to flexible	*transient reference to*

Fig. 2

To illustrate the production of the chart consider the rules for widening.

widened to SIZETY *real* FORM : MEEK SIZETY *integral* FORM.

This says that if one wants to widen to produce a real then the last coercion applied must have been included in the set of MEEK coercions and this must have produced an integer. Thus the only way one can reach widening from an integer to a real must be from the box marked MEEK. If any other routes happen to lead to this they must never produce an integral result.

Similarly for the other rules, e.g.

widened to structured with SIZETY *real field letter r letter e*
SIZETY *real field letter i letter m mode* FORM:
MEEK SIZETY *real* FORM ;
widened to SIZETY *real* FORM.

The first of the above hyperalternatives, i.e. MEEK SIZETY *real* FORM, again indicates the last coercion must have been one of the MEEK coercions. The

second hyperalternative indicates that the last coercion could have been widening to real. Thus the appropriate part of the chart can be produced. The remaining parts of the chart concerned with widening and the other coercions are similar.

With this introduction it should be possible for the reader to inspect each of the rules given in Chapter 6 of the Report and verify that the chart is indeed correct. Note that the metanotions STRONG, FIRM, etc. merely indicate the coercions available in that particular syntactic position.

At this stage one can return to the earlier assignation *x* := "*A*" and note from the chart and therefore the syntax rules that no coercion will allow an object of mode **char** to be coerced, even in a strong position, to an object of mode **real**.

3.3.4. Introducing predicates into the syntax

The next stage in building up the syntax of ALGOL 68 is the introduction of predicates. Throughout the Report there are hypernotions such as

where DYADIC TAD *identified in* NEST
where (ADIC) *is* (MONADIC)
unless (SAFE1) *contains* (*remember* MOID1 MOID2)
  or (SAFE2) *contains* (*remember* MOID2 MOID1)

Each of *identified in*, *is* and *contains* is an example of a predicate. In general in the Report predicates are preceded by either *where* or *unless* and the metarule

WHETHER :: *where* ; *unless*.

allows WHETHER to be used in place of either *where* or *unless.*

3.3.4.1. Using predicates. Predicates resemble booleans in many respects. The way these predicates are manipulated further resembles booleans. Thus one can join together predicates by means of *and* and *or* and thereby set up a kind of predicate algebra. Negation can also be achieved since *unless* serves as a kind of opposite of *where*. Thus it is possible to devise an algebra allowing the manipulation of predicates. (The Report does this in section R1.3.1.)

The mechanism by which predicates are defined in the Report is similar to the mechanism used to define the other hypernotions. Thus one uses the hyperrules starting with WHETHER, *where* or *unless* in conjunction with the metarules and using consistent substitution, etc. one can obtain syntax rules for the predicates and thereby manipulate them.

One could however regard the syntax rules for predicates as yet another grammar, i.e. a grammar for explicitly defining predicates. (But, of course, strictly speaking, it is not ; it is part of the two-level-grammar used for defining ALGOL 68.) If one does regard these syntax rules for predicates as a grammar it

is natural to ask what the terminals of the grammar are. Curiously there are no terminals or, more accurately, the only terminal is EMPTY. To be less vague about this consider an example.

In the rule

> *strong* MOID FORM *coercee*:
> *where* (FORM) *is* (MORF), STRONG MOID MORF;
> *where* (FORM) *is* (COMORF), STRONG MOID COMORF,
> *unless* (STRONG MOID) *is* (*deprocedured to void*).

The symbols (and) merely act as brackets to delimit the "operands" of *is*. In the general case operands can be any sequence of small letters and if *is* itself happened to be present within such sequences one could foresee certain ambiguities arising. Hence the need for (and).

In this rule substitution of metanotions has to be performed consistently in the usual way. Thus, in particular, the metanotions FORM, MORF and COMORF have to be replaced. There are several possibilities that can arise and these are characterized by the examples of substitutions given below. To keep strictly to the Report the metanotion NEST should be included but this will be ignored for the present.

Each of FORM, MORF and COMORF could be replaced by, respectively,

(i)	*slice*	,	*call*	,	*assignation*
(ii)	*assignation*	,	*call*	,	*assignation*
(iii)	*call*	,	*call*	,	*assignation*.

Now *is* is an example of a predicate. Moreover

where (FORM) *is* (MORF)

disappears entirely (as can be seen by chasing through the rules defining *is* in section R1.3.1) if what replaces (FORM) is the same as what replaces (MORF). *is* indicates whether one sequence of small letters is the same as another sequence. Thus taking the above three examples

> *where* (*slice*) *is* (*call*) does not disappear
> *where* (*assignation*) *is* (*call*) does not disappear
> *where* (*call*) *is* (*call*) does disappear

Similarly *where* (FORM) *is* (COMORF) disappears only in example (ii) above.

When *where* (FORM) *is* (MORF) disappears the syntax rule then gives *strong* MOID *call coercee* : STRONG MOID *call* ;

where (*call*) *is* (*assignation*), STRONG MOID *assignation*,
unless (STRONG MOID) *is* (*deprocedured to void*).

The predicate

where (*call*) *is* (*assignation*)

cannot be made to disappear and consequently one can never produce only a sequence of terminal symbols from the second hyperalternative. The predicate is always there. Such hyperalternatives are generally called *blind alleys*. Thus one can obtain only

strong MOID *call coercee* : STRONG MOID *call*.

Consider now the second case, i.e. the case in which FORM and COMORF are replaced by *assignation* and MORF by *call*. Then

where (FORM) *is* (COMORF)

disappears and

where (FORM) *is* (MORF)

does not disappear. Thus one obtains as the only possibility

strong MOID *assignation coercee*:
STRONG MOID *assignation*,
unless (STRONG MOID) *is* (*deprocedured to void*).

Now one has to test the truth of

unless (STRONG MOID) *is* (*deprocedured to void*).

For this to disappear one must verify that

(STRONG MOID) *is* (*deprocedured to void*)

yields *false*. From the earlier section on coercions this is as one would expect. Voiding takes place rather than deproceduring.

Note that in the remaining case i.e. replacing FORM, MORF and COMORF by *slice*, *call* and *assignation* respectively yields nothing of value since both

(FORM) *is* (MORF)

and

(FORM) *is* (COMORF)

yield *false*.

With this introduction one can see the reason for the introduction of predicates. As the Report says (see the remarks at the start of section R1.3) the reason is twofold

(i) predicates are used to enforce certain restrictions on the production trees

(ii) a more modest use is to reduce the number of hyper-rules by grouping several similar cases as alternatives in one rule. In these cases predicates are used to test which alternative applies.

3.3.4.2. Syntax of predicates. From the example it should be apparent that if a predicate is preceded by *where* then it will disappear if one can prove that the predicate yields *true*. If it is preceded by *unless* it will disappear if one can prove the predicate yields *false*. These remarks appear in the syntax in the form of the two rules

where true : EMPTY.
where false: EMPTY.

in conjunction with the metarule

EMPTY ::.

There are no such rules for

where false

or

unless true

and consequently these are what have been termed blind alleys. In fact other blind alleys do exist but these are the more obvious examples.

The rules for combining predicates should now be easily followed.

Example combining predicates

The following illustrate the rules for combining predicates.

(i) *where* THING1 *and* THING2 : *where* THING1, *where* THING2. Thus for this predicate to disappear both must disappear, i.e. both THING1 and THING2 must yield *true*. This is in keeping with what one would expect from boolean algebra.

(ii) *unless* THING1 *and* THING2 : *unless* THING1; *unless* THING2. THING1 *and* THING2 produces *false* if either THING1 or THING2 produces *false* i.e. *unless* THING1 must disappear or *unless* THING2 disappears.

At this stage the meaning of each of the predicates should be investigated. To save the task of going into them all in detail an illustrative example of one predicate is given below. However, in Appendix B a list of the predicates is given together with a brief description of the meaning of each. This should enable the reader to read with some ease and understanding the various rules. If he wishes to determine in more detail the meaning of any particular predicate he can obtain this from the appropriate sections of the Report. In fact the various predicates are named rather imaginatively and remembering their meaning is no hardship.

Example on predicates

The following illustrates the definition of the predicate *is*

(i) *where* (*a*) *is* (*a*) disappears.

This is obtained from the following sequence of rules

where (*a*) *is* (*a*): *where* (*a*) *begins with* (*a*) *and* (*a*) *begins with* (*a*).

where (*a*) *begins with* (*a*) *and* (*a*) *begins with* (*a*):

where (*a*) *begins with* (*a*), *where* (*a*) *begins with* (*a*).

It remains to consider only the one predicate

where (*a*) *begins with* (*a*).

Continuing as before

where (*a*) *begins with* (*a*):

where (*a*) *coincides with* (*a*) *in* (*abcdefghijklmnopqrstuvwxyz*) *and* (EMPTY) *begins with* (EMPTY).

This last predicate requires that both

where (*a*) *coincides with* (*a*) *in* (*abcdefghijklmnopqrstuvwxyz*)

and

where (EMPTY) *begins with* (EMPTY)

disappear. The required result follows in a straightforward manner from the syntax of predicates.

(ii) *where* (*a*) *is* (*b*) does not disappear.

Following the grammar in the same way as above one is lead to the stage where it is necessary to look at

where (*a*) *coincides with* (*b*) *in* (*abcdefghijklmnopqrstuvwxyz*).

This is a blind alley since there is no rule that will lead any further. There is a hyper-rule

unless (ALPHA1) *coincides with* (ALPHA2) *in* (NOTION): . . . . .

but there is no rule of the form

where (ALPHA1) *coincides with* (ALPHA2) *in* (NOTION): . . . . . .

Note that it is not sufficient to have a rule such as

where (ALPHA) *coincides with* (ALPHA) *in* (NOTION): . . . .

since consistent substitution would then allow only the comparison of identical letters. The particular example under consideration requires the comparison of different letters.

(iii) *where* (*aa*) *is* (*a*) does not disappear.

Following the rules leads to, among others, the predicates

where (*a*) *begins with* (*aa*)

and then to

where (EMPTY) *begins with* (*a*).

But this leads to

where false

a blind alley.

3.3.5. Introducing NESTs into the syntax

The final stage of inserting more information into the syntax of ALGOL 68 is just the process of including the metanotion NEST and the information associated with NESTs. The information contained in NESTs is basically the information which the compiler writer must insert into his symbol tables.

In order to understand the motivation for and the meaning of the metanotion NEST, etc. consider the following observations. The declaration

int *x*

is not always a legal declaration in ALGOL 68. Whether or not it is legal depends on the context in which the declaration takes place. In the context

real *x* ; **int** *x*

it would not be legal since *x* has been previously used in an identifier declaration within the same reach. Similarly the assignation

$x := y$

is not always legal. The legality of this depends on whether or not *x* and *y* have been previously declared and whether the modes of *x* and *y* are compatible.

Thus there are context sensitive conditions that have to be satisfied in order that constructions are legal. The motivation for the inclusion of the metanotion NEST lies in the formalization of these context conditions. The information included in NEST, etc. should then indicate that the declaration **int** *x* in the context of

real *x* ; **int** *x*

is illegal. Similarly it should indicate that

$x := y$

is illegal unless suitable declarations appear for *x* and *y*.

In more general terms the NEST information formalizes the process of identifying applied occurrences with their appropriate defining occurrences. This then includes consideration of mode indications, operators, identifiers, labels and field selectors. Moreover the NEST information must reflect in some sense the "block structure" of ALGOL 68 itself.

3.3.5.1. The metanotion TAX. In order to simplify the understanding of what follows some remarks about metanotions and predicates of special interest are included together with some other more general remarks.

The metanotion TAX includes all identifiers and indications of any kind. Thus it includes sequences of symbols used to identify operators, modes, field selectors, etc. The particular metarule is

TAX :: TAG; TAB ; TAD ; TAM.

TAM includes all indications used for monadic operators, TAD includes indications for dyadic operators and TAB includes indications for modes, TAG includes identifiers used for labels (excluding the label symbol :), for field selectors and for mode identifiers.

The metanotions TAD and TAM are often grouped together under the metanotion TAO. Thus TAO encompasses all indications for operators.

3.3.5.2. The metanotion QUALITY. Under QUALITY are grouped a set of characteristics one can associate with the different indicators included under TAX. Thus an identifier used as a label has the QUALITY *label* associated with it. A selector has as its QUALITY one from the set MODE *field*. Thus *integral field* or *reference to character field* are both examples of a QUALITY which one might associate with a field selector. An operator appearing in a priority declaration takes on a QUALITY which consists of the word *priority* followed by a representation of the priority number in Roman numerals.

In summary then

QUALITY	Objects with corresponding *QUALITY are indications given by*	*Example*
MODE	identifier declarations	**real** *x*
MOID TALLY	mode declarations	**mode u** = . . .
DYADIC	priority declarations	**prio** *?* = 3
label	labels	*l* :
MODE field	selectors	*age* **of** *man*

Combining QUALITY and TAX gives QUALITY TAX and this encompasses the various indications and their associated characteristics as given by a particular program. Thus under QUALITY TAX might be included

label l

or

priority iii ?

assuming that in the particular program under discussion there appeared respectively

l:

or

prio *?* = 3

The combination QUALITY TAX is rather important and it is just this that is used in relating applied with defining occurrences. This will be further explained below.

3.3.5.3. The metanotion PROP. For the moment consider the metanotion PROP. PROP encompasses many protonotions all of which are of the form QUALITY TAX. Under PROP are included the metanotions

(i)	DEC	standing for declaration and including identifier, priority, operator and mode declarations
(ii)	LAB	standing for labels
(iii)	FIELD	standing for selector.

To reiterate, each of the protonotions included under DEC, LAB and FIELD is of the form QUALITY TAX. To look at this in more detail

PROP:: DEC; LAB; FIELD.
DEC:: MODE TAG; *priority* PRIO TAD; MOID TALLY TAB; DUO TAD;
MONO TAM.
LAB:: *label* TAG.
FIELD::MODE *field* TAG.

After DEC comes MODE TAG. MODE is included under QUALITY and TAG is included under TAX. Similarly after LAB comes *label* TAG. *label* is included under QUALITY and TAG under TAX. Likewise for the other PROPs. In particular situations then the PROPs can be used to associate a QUALITY with a particular TAX.

This concludes one aspect of the introduction to NESTs. The next part involves consideration at an empirical level of NESTs themselves including how they are constructed, their function, etc.

3.3.5.4. LAYER and NEST. When an ALGOL 68 program is written the writer makes certain assumptions about the environment in which the program is compiled. He assumes, for instance, that the compiler "understands" certain primitive operations such as +, −, ×, /, etc., defined on the integers, reals, etc. together with certain procedures such as *sin, cos, read, print* and so on. The Report refers to such a program as a *particular program*. One can think of such a program as if it were embedded in a clause containing all the standard

declarations together with system libraries and the like and even other programs being executed in parallel. This entire system is what the Report calls a program. So basically a program runs in a null environment or, as the Report calls it, a *primal environment* (which is empty).

The environment in which particular programs (as defined above) are run will be called the *standard environment*. Thus the standard environment contains the declarations of the procedure *sin*, *cos*, *print*, etc., together with declarations of pi, the standard operators, etc. In the course of executing an ALGOL 68 program the environment – or in the terms used in the Report the *environ* – in which the different statements are elaborated will in general be altered by the various declarations. Thus on elaborating a declaration the environ is augmented in the appropriate way and on leaving a clause any local quantities become unavailable. The process of augmenting is described by saying that the environ consists of the old environ together with a *locale*, the locale containing the new declarations declared in that reach. An environ is then made up of a set of ordered locales. Any construction, e.g. an assignation, is elaborated within a particular environ. The pair, the construction with its associated environ, is called a *scene*.

This whole process of creating and destroying environs has to be reflected in the syntax of NESTs. In the syntax a NEST corresponds to an environ and a LAYER corresponds to a locale. The appropriate metarules are

NEST:: LAYER; NEST LAYER.
LAYER:: *new* DECSETY LABSETY.

DECSETY LABSETY provides a (possibly empty) ordered sequence of DECS and/or LABS and as previously noted these are just sequences of QUALITY TAXs with the declarations preceding the labels. Note that in a particular reach declarations must precede defining occurrences of labels; hence DECSETY appears before LABSETY. Note also the presence of *new*. Each occurrence of *new* indicates a new locale.

To take a particular example note that under NEST would be included

new new label letter l label letter h new integer identifier letter i

and this would reflect a program whose skeleton looked like

```
l: . . . .
. . . . . .
h: . . . .
. . . . . .
begin int i=4
    . . . . . . . . . .
```

Here the QUALITY TAXs are just

label letter l
label letter h

and

integer identifier letter i

and these reflect the defining occurrences of the different indicators.

Defining occurrences appear in the Report in the syntax of declarations. There they appear in the form

QUALITY *defining indicator with* TAX

Thus we can say that each QUALITY TAX in an environ corresponds to some such QUALITY *defining indicator with* TAX.

Before proceeding further there are one or two points to note about the example given above. First note the double appearance of *new* at the start of this particular NEST. The primal environment is empty and hence the first *new*. Secondly note that, for simplicity only, no account has been taken of the standard environment. In practice any particular program would take some account of the declarations, etc, contained within the standard environment.

The time has now arrived to look at the inclusion of the context conditions in the syntax. Take defining occurrences first of all.

Defining occurrences of indicators appear in declarations and for the declarations to be legal certain context conditions, some of which have already been briefly mentioned, have to be satisfied. The conditions are contained in section 4.8 of the Report. The first of these caters for indicators excluding field selectors; the second caters for field selectors.

The rule for dealing with the first possibility is

QUALITY NEST *new* PROPSETY1 QUALITY TAX PROPSETY2
defining INDICATOR *with* TAX:
where QUALITY TAX *independent* PROPSETY1 PROPSETY2, TAX *token*.

In this hyperrule note that

NEST *new* PROPSETY1 QUALITY TAX PROPSETY2

is actually a NEST with the most recent locale being represented by the LAYER

new PROPSETY1 QUALITY TAX PROPSETY2.

For field selectors the relevant rule is similar and is

> MODE *field* PROPSETY1 MODE *field* TAG PROPSETY2
> *defining field selector with* TAG:
> *where* MODE *field* TAG *independent* PROPSETY1 PROPSETY 2,
> TAG *token*.

The first rule therefore states that one can add a new QUALITY TAX to the locale provided that QUALITY TAX is independent of the sets PROPSETY1 and PROPSETY2. The predicate *independent* is defined in the usual way. But briefly a sufficient condition for the independence of two QUALITY TAXs is that the TAXs differ. This is also necessary except in the case of some TAX being a TAO, i.e. an operator indicator. In testing the independence of QUALITY1 TAX and QUALITY2 TAX the following hold

an indication used for a mode (i.e. a MOID TALLY) is never independent of any other indication;

an indication used for a monadic operator is always independent of an indication used for a dyadic operator;

any indication used for operator declarations is independent of an indication used in a priority declaration;

when two indications are both used for either monadic or dyadic operators then independence occurs unless the corresponding parameters are firmly related.

It is of interest to note that the rule for field selectors originates from the rule

> NEST MODE FIELDS *definition of* MODE *field* TAG:
> MODE *field* FIELDS *defining field selector with* TAG.

Note the disappearance of NEST on the right hand side of this rule. Thus it does not matter if the particular selectors clash with identifiers already used in that reach for some other purpose. The earlier rule implies that it is necessary only to check that this particular field selector is not the same as another selector for the same structure.

Now consider applied occurrences. As indicated earlier applied occurrences require that there is a corresponding defining occurrence. Thus

$$x{:=}a+b\times c$$

is legal only if suitable declarations exist for x, a, b and c. The environ information is passed through the syntax by means of the metanotion NEST. The complete hyperrule for an assignation reads

> REF *to* MODE NEST *assignation*:
> REF *to* MODE NEST *destination*, *becomes token*,
> MODE NEST *source*.

Notice the way in which the NEST information is passed through the syntax. This is further exemplified in the rules

REF *to* MODE NEST *destination*:
soft REF *to* MODE NEST TERTIARY.
MODE1 NEST *source* : *strong* MODE2 NEST *unit*,
where MODE1 *deflexes to* MODE2.

In the particular example under consideration the source is a

strong MODE2 NEST ADIC *formula*.

In the syntax for a formula the NEST information is again passed through so that one obtains a

MODE NEST *operand*.

Chasing through the syntax for this eventually leads in our case to

QUALITY NEST *applied* INDICATOR *with* TAX.

Each of *a*, *b* and *c* as used in the above formula qualify for this description.

The syntax for applied occurrences again involves two rules. The first of these rules deals with indicators other than field selectors. The second takes care of field selectors. The rule dealing with the first possibility is

QUALITY NEST *applied* INDICATOR *with* TAX:
where QUALITY TAX *identified in* NEST, TAX *token*.

Thus the legality of an applied occurrence depends on whether the appropriate QUALITY TAX can be identified in the current NEST. Here *identified in* is used in a technical sense and refers to a predicate which is defined in R7.2.1a. This predicate is defined in terms of another predicate, viz *resides in*, which is used in defining the legality of an applied occurrence of a field selector. Before looking at these predicates note that the second rule is

MODE *field* FIELDS *applied field selector with* TAG:
where MODE *field* TAG *resides in* FIELDS, TAG *token*.

The predicate *identified in* asks whether a particular PROP (i.e. QUALITY TAX) can be identified in a NEST. Since a NEST is made up of LAYERs the predicate is defined recursively and therefore searches through the different LAYERs. Asking whether the PROP is in a particular LAYER is performed by

the predicate *resides in*. This predicate, as noted above, is used also in the syntax of field selectors.

The predicate *resides in* has to test if one QUALITY TAX resides in a sequence of QUALITY TAXs (i.e. PROPs). This is again performed recursively and eventually one is left with the problem of comparing two QUALITY TAXs. For labels or dyadic operator indication a simple match is involved. In other cases it is necessary to test the equivalence of two modes.

This then completes the investigation into defining and applied occurrences, i.e. into LAYERs and NESTs. It should be remembered that eventually there is a standard environment at the outmost level. The syntax itself contains no mention of the standard environment. Indeed the programmer himself can design his own libraries and thereby create his own environment in which to run his programs. The syntax itself makes use of just LAYER and NEST as illustrated previously. In this way the syntax encourages the use of precompiled segments, these corresponding to outer layers.

3.3.6. Miscellaneous examples

Before leaving the syntax of ALGOL 68 it is of interest to take some constructions and look at their syntax. Three different examples will be considered:

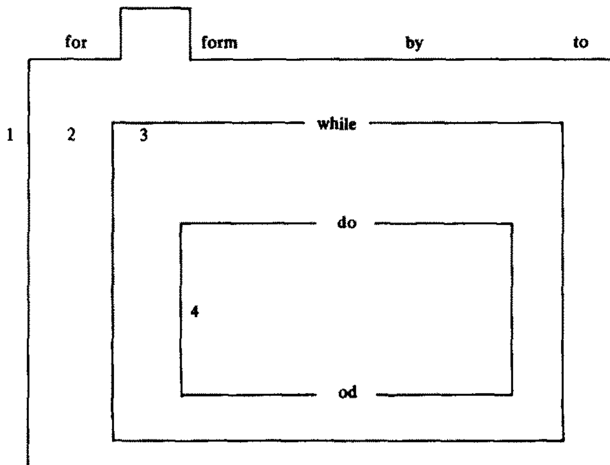
(a) the syntax for loop clauses
 this syntax will illustrate the manner in which NESTs are created

(b) the syntax of formulae

(c) the syntax explaining the equivalence and the well-formedness of modes.

These will now be considered in turn.

3.3.6.1. The syntax of loop clauses. The interesting part of the syntax of loop-clauses is the construction of the various environs. This is just what will be considered below. The Report displays the loop clause thus



The integers 1,2,3,4 indicate the environs to be represented in the syntax by NEST1, NEST2, NEST3 and NEST4 respectively. NEST1 is the environ associated with the loop clause itself. NEST2 contains the information in NEST1 together with the integer identifier declared after the **for**. NEST3 consists of NEST2 plus any declarations appearing between the **while** and the **do**. Finally between the **do** and **od** one can use any of the information in NEST3 together with any declarations occurring between **do** and **od**. This is just NEST4.

The rule for a loop clause reads

strong void NEST1 *loop clause*:
NEST1 STYLE *for part defining new integral* TAG2,
NEST1 STYLE *intervals*,
NEST1 STYLE *repeating part with integral* TAG2.

Now consider

NEST1 STYLE *repeating part with* DEC2:
NEST1 *new* DEC2 STYLE *while do part*;
NEST1 *new* DEC2 STYLE *do part*.

This then indicates that the environ associated with the *while do part* (or just the *do part* if the *while part* has been omitted) is NEST1 with the new locale consisting of the *integral* TAG2. This environ is effectively NEST2.

Assume now that the *while do part* does exist. Then from

NEST2 STYLE *while do part* : NEST2 STYLE *while part defining* LAYER3,
NEST2 LAYER3 STYLE *do part*.

NEST2 LAYER3 is effectively NEST3 since to NEST2 is added the declarations contained within the *while part*.

Finally

NEST3 STYLE *do part* : STYLE *do token*,
strong void NEST3 *serial clause defining* LAYER4,
STYLE *od token*.

NEST4 is thereby produced.

3.3.6.2. The syntax of formula. Some explanatory remarks will show that this syntax is remarkably simple. Yet it is rather sophisticated for the syntax of formulae (just as with other constructs) has to determine implied bracketings. In particular it has to determine implied bracketings of expressions as various as

$a + b \times c \uparrow n - e$
$a + b + c - d - e$
$a \uparrow b \uparrow c$
age **of** b [i] $+ c \times d$

and so on.

With formulae (just as with other ALGOL 68 constructions) one associates certain information such as a value, a mode and a NEST. But unlike other constructions one also associates a priority. If all extraneous brackets are removed the priority is just that of the lowest priority operator not enclosed in brackets. To give some examples (using the standard operators)

$a + b \times c$ is a priority 6 formula since + has priority 6

$a \times b \uparrow c$ is a priority 7 formula since X has priority 7

$-(x + y \times 7)$ is a priority 10 formula since monadic minus has priority 10.

Note the similarity with formulae as contained in the formal definition of ALGOL W.

Priorities appear in this syntax in several forms:

DYADIC covers priority 1, priority 2, ... , priority 9 i.e. the priorities of dyadic operators;

MONADIC is the priority of all monadic operators i.e. effectively priority 10;

DYADIC TALLY gives a priority (strictly) greater than that of DYADIC;

DYADIC TALLETY gives a priority greater than or equal to that of DYADIC.

Note also that operators themselves appear in this syntax as procedures. A dyadic operator since it has two operands appears as

procedure with MODE1 *parameter* MODE2 *parameter*
yielding MOID NEST *applied operator with* TAD.

MOID1 must be the mode associated with the left hand operand and MOID2 is the mode associated with the right hand operand. The result is of mode MOID and it is therefore a MOID formula. Monadic operators appear as

procedure with MODE *parameter yielding* MOID NEST
applied operator with TAM

and similar remarks about modes of operands and results apply.

The syntax rule for a formula reads

MOID NEST DYADIC *formula*:
MODE1 NEST DYADIC TALLETY *operand*,
procedure with MODE1 *parameter* MODE2 *parameter*
yielding MOID NEST *applied operator with* TAD,
where DYADIC TAD *identified in* NEST,
MODE2 NEST DYADIC TALLY *operand*.

The right hand operand is a formula with strictly higher priority than the formula itself. The left hand operand is also a formula whose priority is greater than, but possibly equal to, that of the formula itself. This has substantial implications as far as implied bracketing or parsing is concerned.

$$a + b \times c \uparrow d$$

must be bracketed as

$$a + (b \times (c \uparrow d)).$$

Any other parsing would contradict the fact that the right hand operand should have a higher priority than the formula being considered. In the formula

$$a - b - c$$

the bracketing must be

$$(a - b) - c$$

and so on. In

age **of** *b* [*i*] + *c* × *d*

it is necessary to invoke that part of the grammar defining UNIT, TERTIARY and so on. But the implied bracketing is

(*age* **of** (*b* [*i*])) + (*c* × *d*).

3.3.6.3. The equivalence and the well-formedness of modes. The intention of this section is to give some background information to that part of the Report dealing with the equivalence and the shielding of modes. An illustrative example is also included. (See Chapter 7 of the Report.)

The problem of determining the equivalence of modes is just the problem of determining whether the same mode has been written in different ways. The following are different ways of spelling the same mode

(i) **union (int, real)** and **union (real, int)**
(ii) **union (int, real, bool)** and **union (bool, int,real)**
(iii) the modes **a** and **b** defined as
mode a = struct (ref a *b*)
mode b = struct (ref struct (ref b *b*) *b*)

(This last example is taken from the Report.)

To understand the problem of determining whether two modes are equivalent it is necessary to understand the way in which recursive (or mutually recursive) modes are defined in the syntax.

Consider the declarations of the modes **a** and **b** given above. **a** is defined by

mode a = struct (ref a *b***)**

The first occurrence of **a** above can be regarded as a defining occurrence and the second occurrence is an applied occurrence. The **a** itself does not appear in the mode – in fact **a** acts as a dummy – but instead is replaced by a derivative of MU, *mu*TALLY. Thus one could write the above mode as

mui definition of structured with reference to mui application field letter b mode.

Similarly the mode **b** defined by

mode b = struct (ref struct (ref b *b***)** *b***)**

might appear in the syntax as

muii definition of structured with reference to structured with reference to muii application field letter b mode field letter b mode.

The syntax for testing equivalence must deal with these two modes. It does this by representing a mode as a tree structure and if the modes are recursive the tree structure is continually expanded until either the trees are shown to be equivalent or they are shown to be different. However it is better for the reader to work through some examples in order to get a feeling for precisely what the rules do. To help him in this an example is given later in this section. The reader may note that the rule for proving the equivalence of two modes is really an application of proof by recursive induction.

Modes such as **d** declared by

mode d = [1:10] d

are certainly not well-formed. Roughly speaking a mode **x** declared recursively as in

mode x =....... x

is well-formed if the defining occurrence and any applied occurrence are separated by one *yin*-HEAD and one *yang*-HEAD. The *yin* and *yang*-HEAD may be the same HEAD e.g. *procedure with*, i.e. with parameters.

The following quotation regarding *yin* and *yang* comes from The New Encyclopaedia Britannica [19].

> yin–yang, in Chinese thought, the two complementary forces, or principles, that make up all aspects and phenomena of life. Yin is conceived of as Earth, female, dark, passive and

absorbing; it is present in even numbers, in valleys and streams, and is represented by the tiger, the colour orange, and a broken line. Yang is conceived of as Heaven, male, light, active and penetrating; it is present in odd numbers, in mountains and is represented by the dragon, the colour azure, and an unbroken line. The two are both said to proceed from the Supreme Ultimate (T'ai Chi), their interplay on one another (as one increases the other decreases) being a description of the actual process of the universe and all that is in it. In harmony the two are depicted as the light and dark half of a circle.

The classification of HEADs (note that not all HEADs are included) into *yin*-heads and *yang*-heads is given below

yin-HEADs	*yang*-HEADs
procedure with	*procedure with*
reference to	*structured with*
transient reference to	
procedure yielding	

The idea behind these restrictions is two-fold. In the first place an object of a particular mode must occupy a finite amount of space (not an infinite amount). This rule therefore excludes mode a declared as

mode a = **struct** (**a** *c*, **char** *d*)

The second rule is designed to prevent (syntactic) ambiguities resulting from coercion. Note that if mode **c** and **d** defined as in

mode c = **ref c**, **d** = **proc d**

were allowed then dereferencing an object of mode **c** would produce a result of mode **c** and deproceduring an object of mode **d** would yield a result of mode **d**.

It was remarked earlier that the modes a and **b** defined by

mode a = **struct** (**ref a** *b*)

and

mode b = **struct** (**ref struct** (**ref b** *b*)*b*)

are equivalent. The process of following the relevant rules is now undertaken. SAFE will be used many times and eventually becomes rather complex. To aid the reader in understanding the SAFEs a diagram is given at the end of this section. It indicates the information stored in the SAFEs at different stages of the evaluation.

Let MOID1 denoted mode **a**. This appears as

mui definition of structured with reference to
mui application field letter b mode.

Let MOID2 denote mode **b** i.e.

> *muii definition of structured with reference to*
> *structured with reference to muii application*
> *field letter b mode field letter b mode.*

Three syntax rules R7.3.1a,b,c will be required in this exercise. For completeness these are \

7.3.1a WHETHER MOID1 *equivalent* MOID2:
  WHETHER *safe* MOID1 *equivalent safe* MOID2.

7.3.1b WHETHER SAFE1 MOID1 *equivalent* SAFE2 MOID2:
  where (SAFE1) *contains* (*remember* MOID1 MOID2)
    or(SAFE2) *contains* (*remember* MOID2 MOID1),
    WHETHER *true*;
  unless (SAFE1) *contains* (*remember* MOID1 MOID2)
    or (SAFE2) *contains* (*remember* MOID2 MOID1),
    WHETHER (HEAD3) *is* (HEAD4)
    and remember MOID1 MOID2 SAFE3 TAILETY3
    equivalent SAFE4 TAILETY4,
    where SAFE3 HEAD3 TAILETY3 *develops from* SAFE1 MOID1
    and SAFE4 HEAD4 TAILETY4 *develops from* SAFE2 MOID2.

7.3.1c WHETHER SAFE2 HEAD TAILETY *develops from* SAFE1 MOID:
  where (MOID) *is* (HEAD TAILETY),
    WHETHER (HEAD) *shields* SAFE1 *to* SAFE2;
  where (MOID) *is* (MU *definition of* MODE),
    unless (SAFE1) *contains* (MU *has*),
    WHETHER SAFE2 HEAD TAILETY *develops from*
    MU *has* MODE SAFE1 MODE;
  where (MOID) *is* (MU *application*)
    and (SAFE1) *is* (NOTION MU *has* MODE SAFE3)
    and (NOTION) *contains* (*yin*) *and* (NOTION) *contains* (*yang*),
    WHETHER SAFE2 HEAD TAILETY *develops from* SAFE1 MODE.

Rule R7.3.1a says that testing the equivalence of MOID1 and MOID2 amounts to testing the equivalence of *safe* MOID1 and *safe* MOID2. Now apply R7.3.1b with SAFE1 as *safe* and SAFE2 as *safe*. This tells us to evaluate

  WHETHER (HEAD3) *is* (HEAD4)
  and remember MOID1 MOID2 SAFE3 TAILETY3
    equivalent SAFE4 TAILETY4,
  where SAFE3 HEAD3 TAILETY3 *develops from* SAFE1 MOID1
  and SAFE4 HEAD4 TAILETY4 *develops from* SAFE2 MOID2.

This requires us to determine SAFE3, HEAD3 and TAILETY3 as well as SAFE4, HEAD4 and TAILETY4. These can be determined from R7.3.1c. Since determining SAFE3, etc. requires a similar process to determining SAFE4 it will suffice to consider SAFE3, etc.

From R7.3.1c

where SAFE3 HEAD3 TAILETY3 *develops from* SAFE1 MOID1 yields
where SAFE3 HEAD3 TAILETY3 *develops from*
mui has MODE1 SAFE1 MODE1

and here MODE1 denotes

structured with reference to mui application field letter b mode.

Another application of R7.3.1c yields

HEAD3	must be *structured with*
TAILETY3	must be *reference to mui application field letter b mode*

and using, R7.4.1c, WHETHER (*structured with*) *shields* SAFE *to yang* SAFE: WHETHER *true*.

SAFE3	must be *yang mui has* MODE1 SAFE1.

A similar approach produces SAFE4, etc. Thus

HEAD4	must be *structured with*
TAILETY4	must be *reference to structured with reference to muii application field letter b mode field letter b mode*

and

SAFE4	must be *yang muii has* MODE2 SAFE2

where now MODE2 is

structured with reference to structured with reference to muii application field letter b mode field letter b mode.

It is now possible to return to the point reached before it was necessary to side-track to find SAFE3, SAFE4, etc. Note that

where (HEAD3) *is* (HEAD4)

is easily evaluated. It now remains to evaluate

where remember MOID1 MOID2 SAFE3 TAILETY3
equivalent SAFE4 TAILETY4.

This involves comparing the different fields for equivalence. *mode* is removed from the end of both TAILETY3 and TAILETY4 and this gives, since the TAGs associated with TAILETY3 and TAILETY4 are both *b*,

where remember MOID1 MOID2 SAFE3 MODE5
equivalent SAFE4 MODE6.

Here MODE5 and MODE6 denote, respectively,

reference to mui application

and

reference to structured with reference to muii application field letter b mode.

At this stage it is convenient to recap and see just what has been achieved. It is now necessary to evaluate

where remember MOID1 MOID2 SAFE3 MODE5
equivalent SAFE4 MODE6.

It will be more convenient to rewrite this as

where SAFE5 MODE5 *equivalent* SAFE6 MODE6

where

MODE5 is *reference to mui application*
MODE6 is *reference to structured with reference to muii application field letter b mode*

and noting that MOID1 and MOID2 represent the two original modes that had to be shown equivalent

SAFE5 is
remember MOID1 MOID2 *yang mui has structured with reference to mui application field letter b mode safe.*
SAFE6 is
yang muii has structured with reference to structured with reference to muii application field letter b mode field letter b mode safe.

The evaluation of this new predicate requires a further application of R7.3.1b. This yields

where (HEAD7) *is* (HEAD8) *and*
remember MODE5 MODE6 SAFE7 TAILETY7
equivalent SAFE8 TAILETY8,
where SAFE7 HEAD7 TAILETY7 *develops from* SAFE5 MODE5
and SAFE8 HEAD8 TAILETY8 *develops from* SAFE6 MODE6.

By following a similar approach to that adopted earlier this gives

HEAD7	is *reference to* and HEAD8 is *reference to*
TAILETY7	is *mui application*
TAILETY8	is *structured with reference to muii application field letter b mode*
SAFE7	is *yin* SAFE5
SAFE8	is *yin* SAFE6

It is now necessary to evaluate

where remember MODE5 MODE6 *yin* SAFE5 *mui application*
equivalent yin SAFE6 *structured with reference to*
muii application field letter b mode.

Rewrite this as

where SAFE9 *mui application equivalent* SAFE10
structured with reference to muii
application field letter b mode.

Going back to rule R7.3.1b yet again the above predicate yields the same as

where (HEAD11) *is* (HEAD12)
and remember mui application structured with reference to muii
application field letter b mode SAFE11 TAILETY11
equivalent
SAFE12 TAILETY12
where SAFE11 HEAD11 TAILETY11 *develops from* SAFE9 *mui application*
and SAFE12 HEAD12 TAILETY12 *develops from* SAFE10 *structured*
with reference to muii application field letter b mode.

A treatment similar to that employed earlier will take care of SAFE12 etc. and will give

HEAD12 is *structured with*
TAILETY12 is *reference to muii application field letter b mode*

and

SAFE12 is *yang* SAFE10.

The treatment required for SAFE11 etc. is new. The predicate to be considered is

where SAFE11 HEAD11 TAILETY11 *develops from* SAFE9 *mui application*

and SAFE9 is an abbreviation for

remember
reference to mui application
reference to structured with reference to muii application field letter b mode
yin
remember
MOID1
MOID2
yang mui has structured with reference to mui application field letter b mode safe

MOID1 and MOID2 being the two modes to be shown equivalent in the first place. The last hyperalternative in rule R7.3.1c applies and gives

HEAD11 is *structured with*
TAILETY11 is *reference to mui application field letter b mode*

and

SAFE11 is *yang* SAFE9

The problem now reduces to an evaluation of

where remember muii application structured with reference to muii application
reference to mui application field letter b mode
equivalent
yang SAFE10 *reference to muii application field letter b mode*

Rewrite this as

> *where*
> SAFE13 *reference to mui application field letter b mode*
> *equivalent*
> SAFE14 *reference to muii application field letter b mode.*

This is in turn the same as

> *where*
> SAFE15 *mui application field letter b mode*
> *equivalent*
> SAFE16 *muii application field letter b mode.*
> SAFE15 being just
> *remember*
> *reference to mui application field letter b mode*
> *reference to muii application field letter b mode*
> *yin*
> SAFE13

and SAFE16 is *yin* SAFE14.

Further application now indicates that the predicate yields the same as

> *where*
> SAFE17 *reference to mui application field letter b mode*
> *equivalent*
> SAFE18 *reference to structured with reference to*
> *muii application field letter b mode field letter b mode*

and here SAFE17 is

> *remember*
> *mui application field letter b mode*
> *muii application field letter b mode*
> *yang*
> SAFE15

and SAFE18 is *yang* SAFE16.

The required result will now soon follow a single application of R7.3.1b. Note that in the usual way one can drop the *mode* and the TAG since the two TAGs are identical. The problem reduces to considering the predicate

SAFEs occurring in worked example
(SAFEs grow upwards)

	SAFE17	SAFE18	
remember *mui application field letter b mode* *muii application field letter b mode* *yang*	SAFE15	SAFE16	*yang*
remember *reference to mui application field letter b mode* *reference to muii application field letter b mode* *yin*	SAFE13	SAFE14	*yin*
remember *mui application* *structured with reference to muii application field letter b mode.*	SAFE11	SAFE12	
yang	SAFE9	SAFE10	*yang*
remember *reference to mui application* *reference to structured with* *reference to muii application field letter b mode.*	SAFE7	SAFE8	
yin	SAFE5	SAFE6	*yin*
remember MOID1 MOID2	SAFE3	SAFE4	
yang *mui has structured with reference to mui application field letter b mode*	SAFE1	SAFE2	*yang* *muii has structured with reference to structured with reference to muii application field letter b mode field letter b mode.*
safe			*safe*

where
SAFE17 *reference to mui application*
equivalent
SAFE18 *reference to structured with reference to muii application field letter b mode.*

Now

where (SAFE17) *contains*
(*remember*
reference to mui application
reference to structured with
reference to muii application field
letter b mode)

yields *where true*. Hence the required result follows from R7.3.1b.

3.4. THE SEMANTICS OF ALGOL 68

The meaning to be attached to different constructions and programs is defined in the semantics of the Report. By 'program' is meant the construction defined by the syntax of the Report. Thus it is not a particular program as such i.e. a program written by a user, but it is a particular program together with the standard environment, libraries and so on.

The method of describing the semantics of ALGOL 68 is just English, but English used in a rather formalized way. In describing the semantics many of the words – for example words such as name, value, action, etc. – are used in a technical sense and their precise meaning is therefore defined in an earlier part of the Report. Indeed many of the techniques of programming have been applied to the writing of the semantics. As Sintzoff points out (see [13]) one can notice the use of quantifiers, predicates, variables and recursive functions in the semantics. Moreover the Report is written using top-down structured programming techniques. Even "jumps" do not appear but rather there are conditional and case statements.

To describe precisely the effects of the different programs the Report introduces a hypothetical computer. It then defines in terms of this computer the various actions that have to be taken when the computer elaborates a particular construct.

Since the notion program allows the existence of programs which loop indefinitely – for instance the program

do skip od

loops indefinitely – and programs which require an infinite amount of store – for instance a list could be enlarged indefinitely by an infinite loop – this

hypothetical computer is idealized in some senses. Thus it can be assumed to have an arbitrary large store and it can keep going for an indefinite length of time. Further it can hold integers, reals, etc. of an arbitrary magnitude.

An implementation of ALGOL 68 is then nothing more than a model of the hypothetical computer. The model is assumed to make use of some physical computer and therefore to have certain physical limitations such as limited amounts of store and limits to the size of integers that can be held.

3.4.1. Internal and external objects

To illustrate the formalized approach to the semantics consider *objects* (as described in R2.1.1). The Report goes to some lengths to distinguish between internal and external objects. External objects correspond to some text which appears in a program. Within the hypothetical computer however these external objects are realized in terms of internal objects – they are internal to the hypothetical computer. Included under internal objects are values, locales, environs and scenes.

The values have associated with them a mode which determines the way in which values of that mode can be manipulated. The values can be classified as plain values, names, multiple values, structured values and routines. Plain values include integers, real number, characters, booleans and void values.

It is of interest to see that the Report demands that it should be possible to perform certain operations on certain pairs of numbers and this should never result in an error condition such as overflow. In particular it must be possible to elaborate $n < m$ where n and m are of mode **int** or mode **real**. Note that overflow could result from evaluating $n-m$ and testing for a negative result.

Another kind of internal object is the locale. A locale corresponds to declarations contained in a particular reach. Each reach then has a corresponding locale (which may be empty if there are no declarations). Nested locales will then be built up in a manner which reflects the structure of a particular program. Such a set of locales is called an environ, another kind of internal object.

The final kind of internal object is the scene. Scenes include the constructs in the language, e.g. loop clauses, closed clauses, assignations, etc. together with all the context sensitive information needed for their elaboration. The context sensitive information is contained in an environ. A scene consists of a construct and an environ.

The question of how a construct is represented in the computer now arises. In fact it is held as a tree, a production tree. The production tree is, more or less, the usual syntax tree.

Example on internal and external objects

(i) The external object 123 would have as a corresponding internal object a sequence of pulses representing the bit pattern for 123

(ii) The external object n declared in

int n:=4

would have as its corresponding internal object a name i.e. a value referring to the integer value 4

(iii) the assignation

$$n{:=}4$$

would be represented by a scene consisting of

a production tree representing the assignation

and

an environ containing the necessary context sensitive information.

3.4.2. The hypothetical computer

The process of elaborating scenes involves the hypothetical computer in certain actions. In more usual terminology the computer works by performing actions. These are serial actions, collateral actions and parallel actions corresponding to the elaboration of serial clauses, collateral clauses and parallel clauses. Actions can moreover be expressed in terms of other actions and these in turn can be decomposed into more primitive actions. Eventually one comes across inseparable actions which cannot be so decomposed.

The actions performed by the hypothetical computer take one of three forms:

(i) relationships can be made to hold
(ii) new names can be generated
(iii) other scenes can be elaborated

Only (i) above requires further explanation.

Relationships can be permanent, i.e. independent of a program and its elaboration, or they can be made to hold or cease to hold as a result of actions. The permanent relationships are:

to be of the same mode as
to be smaller than
to be widenable to
to be lengthenable to
to be equivalent to.

If a permanent relationship holds between two values then it will always hold between those values. If it does not hold then it will never hold.

On the other hand there are relationships that can be made to hold as a result of actions. These include:

to be the yield of
to access
to refer to
to be newer (older, the same as) in scope
to be a subname of.

The reader is referred to Chapter 2 of the Report for a description of the meaning of the various relationships.

3.4.3. Transput

Transput, i.e. input and output, is described in terms of ALGOL 68 itself. The various procedures for printing, reading, etc. are all declared in terms of more primitive operations. Even files, channels and books appear there as structures whose definition is included. These structures cannot be examined by the usual method of selection and are therefore special.

The reader is left to read the transput section.

4. CONCLUSION

In this paper we have demonstrated the progression in the formal definition of programming languages from ALGOL 60 through to the revised version of ALGOL 68. Most of the criticisms levelled at the formal definition of ALGOL 60 and ALGOL W have been removed. But perhaps the reader might think that more serious criticisms could be levelled at the formal definition of ALGOL 68. He might think for instance that the Revised ALGOL 68 Report is too complicated and even unreadable.

It should be remembered that the Report did attempt to introduce new ideas into the formal definition of programming languages. It was never intended that the Report should be a document from which one should learn ALGOL 68. The Report should be regarded as a piece of mathematics or logic. To quote

> "The Group wishes to contribute to the solution of the problems of describing a language clearly and completely. The method adopted in this Report is based upon a formalized two-level grammar, with the semantics expressed in natural language, but making use of some carefully and precisely defined terms and concepts. It is recognised, however, that this method may be difficult for the uninitiated reader."

One may or may not agree with the approach adopted by the authors of the ALGOL 68 Report. To some extent this will depend on what one sees as the purpose of the formal definition of a programming language. And one's attitude to this may depend on one's position in the computing community. See, however, [4].

Perhaps this article should have started with an attempt to discuss the role that the formal definition of a programming language should play.

APPENDIX A THE METANOTIONS

This appendix contains a list of metanotions together with some explanation as to their meaning and the reason for their existence.

1. ABC(942L)::*a*;*b*;*c*; . . . ;*z*.

   ABC denotes any small letter.

2. ADIC(542C)::DYADIC;MONADIC.

   Includes priority of all operators. DYADIC gives priorities of dyadic operators and MONADIC gives priority of monadic operators. One generally talks about ADIC formulae where ADIC gives essentially the priority of the lowest priority operator.

3. ALPHA(13B)::*a*;*b*;*c*; . . . ;*z*.

   ALPHA stands for any small letter.

4. BECOMESETY(942J)::*cum becomes*;*cum assigns to*; EMPTY.

   Used for construction of operators. := or =: can be added to operators constructed from the MONADs and NOMADs.

5. BITS(65A)::*structured with row of boolean field* SITHETY *letter aleph mode.*

   Includes **bits** with all possibilities of **long** and **short** included in the metanotion SITHETY.

   (See note one at the end of this appendix)

6. BYTES(65B)::*structured with row of character field* SITHETY *letter aleph mode.*

   Includes **bytes** with all possibilities of **long** and **short** included in metanotion SITHETY.

   (See note one at the end of this appendix)

7. CASE(34B)::*choice using integral*;*choice using* UNITED.

   Includes both kinds of **case** clauses, i.e. selection using integers or modes.

8. CHOICE(34A)::*choice using boolean*;CASE.

   Includes all kinds of conditionals i.e. choices using booleans, integers or united modes (the latter are included under CASE).

9. COLLECTION(A341C)::*union of* PICTURE COLLITEM *mode.*

   Used in transput

   mode collection = union (picture, collitem)

10. COLLITEM(A341D)::*structured with* INSERTION *field letter i digit one procedure yielding integral field letter r letter e letter p integral field letter p* INSERTION *field letter i digit two mode.*

Used in transput

mode collitem = **struct** (**insertion** *i1*, **proc int** *rep*, **int** *p*, **insertion** *i2*)

11. COMARK(A341N)::*zero*; *digit*; *character*.

    All frames included under COMARK can be replicated.

12. COMMON(41A)::*mode*;*priority*;MODINE *identity*;*reference to* MODINE *variable*; MODINE *operation*;PARAMETER;MODE FIELDS.

    In different ALGOL 68 constructions there can be a mode, the operator symbol **op**, or various other possibilities followed by indicators separated by commas. Examples include

int *num*, *den*	in **struct** (**int** *num*, *den*)
real *x*,*y*	in (**real** *x*,*y*):*x* + *y*
	i.e. in procedure declaration

    mode i = **int**, **b** = **bool** in mode declaration
    prio + = 6, – = 6, ... in priority declaration etc.

    COMMON is intended to encompass all these possible forms of declarations.

13. COMORF(61G)::NEST *assignation*;NEST *identity relation*;NEST LEAP *generator*;NEST *cast*;NEST *denoter*;NEST *format text*.

    The set of FORMs is divided into MORFs and COMORFs. The division arises because with FORMs yielding a result of mode NONPROC voiding can be done by deproceduring or voiding. (See note 2 of this Appendix.)

14. CPATTERN(A341I)::*structured with* INSERTION *field letter i integral field letter t letter y letter p letter e row of* INSERTION *field letter c mode*.

    Used in transput

    mode cpattern = **struct** (**insertion** *i*, **int** *type*, **flex** [1:0] **insertion** *c*)

15. DEC(123E)::MODE TAG;*priority* PRIO TAD;MOID TALLY TAB;DUO TAD; MONO TAM.

    Includes all kinds of QUALITY TAX associated with declarations (excluding labels)

    MODE TAG – declaration of identifiers
    priority PRIO TAD – priority declaration
    MOID TALLY TAB – mode declarations
    DUO TAD – dyadic operator declaration
    MONO TAM – monadic operator declarations.

    These are used in the syntax of NESTs and LAYERs. DEC is an abbreviation for declaration

16. DECS(123D)::DEC;DECS DEC.

    A sequence of n ($n \geqslant 1$) QUALITY TAXs of any kind excluding label declarations. (See 14 above).

17. DECSETY(123C)::DECS;EMPTY.

    A possible empty set of DECs

18. DEFIED(48B)::*defining*;*applied*.

Occurrences of identifiers can be either defining or applied. DEFIED encompasses both ideas.

19. DIGIT(942C)::*digit zero;digit one;... ;digit nine.*

    DIGIT stands for any digit; used in, for example, priority declarations.

20. DOP(942M)::DYAD;DYAD *cum* NOMAD.

    Encompasses all the operator symbols for dyadic operators, excluding operators terminated by := or =: and operators represented by bold TAGs.

21. DUO(123H)::*procedure with* PARAMETER1 PARAMETER2 *yielding* MOID.

    Encompasses modes associated with dyadic operators. The left hand operand has mode encompassed by PARAMETER1, the right hand operand has mode encompassed by PARAMETER2 and the result produced by the operator has mode MOID. To be precise operators possess routines of mode DUO.

22. DYAD(942G)::MONAD;NOMAD.

    Encompasses all single operator symbols (ignoring bold TAGs). The subset MONAD are the operators which can appear monadically, the NOMADs cannot.

    (See note 3 at the end of the appendix).

23. DYADIC(542A):*priority* PRIO.

    Includes *priority i* to *priority iii iii iii* and hence covers priorities of all dyadic operators. Formulae have priorities associated with them namely the priority of essentially the operator of lowest priority which is not enclosed in brackets. DYADIC is therefore a kind of adjective that can be applied to formulae.

24. EMPTY(12G)::.

25. ENCLOSED(122A)::*closed;collateral;parallel*,CHOICE;*loop*.

    ENCLOSED encompasses these types of closed clause which can never deliver a result which is then coerced. Ignoring jumps, skips and nihils these are the only units which are not coercees. Particular programs are ENCLOSED clauses.

26. EXTERNAL(A1A)::*standard;library;system;particular.*

    Includes all the different kinds of prelude within which a particular program will run.

27. FIELD(12J)::MODE *field* TAG.

    mode-identifier pair used in forming structures.

28. FIELDS(12I)::FIELD;FIELDS FIELD.

    Encompasses all sequences of mode-identifier pairs as found in structures.

29. FIRM(61B)::MEEK;*united to.*

    Includes all coercions available in FIRM syntactic position i.e. uniting, dereferencing and deproceduring.

30. FIVMAT(A341L)::*mui definition of structured with row of structured with integral field letter c letter p integral field letter c letter o letter u*

letter n letter t integral field letter b letter p row of union of structured with union of PATTERN CPATTERN *structured with* INSERTION *field letter i procedure yielding mui application field letter p letter f mode* GPATTERN *void mode field letter p* INSERTION *field letter i mode* COLLITEM *mode field letter c mode field letter aleph mode.*

Mode used in transput and is equivalent to mode **format**.

31. FLEXETY(12K)::*flexible*,EMPTY.

A multiple value may be referred to by either

(i) *a flexible name* of mode *reference to flexible* ROWS *of* . . . .

or

(ii) *a fixed name* of mode *reference to* ROWS of . . .

FLEXETY therefore includes both possibilities.

32. FORM(61E)::MORF;COMORF.

Includes all cases of unitary clauses with the exception of jumps, skips and nihils. These 3 exceptions do not have any a priori mode associated with them. All objects included under FORM do have an a priori mode.

33. FORMAT(A341A)::*structured with row of* PIECE *field letter aleph mode.*

Used in transput.

mode format = **struct** (**flex** [1:0] **piece** $\mathcal{K}$)

34. FPATTERN(A341J)::*structured with* INSERTION*field letter i procedure yielding* FIVMAT *field letter p letter f mode.*

Used in transput for format patterns.

mode fpattern = **struct** (**insertion** *i*, **proc fivmat** *pf*)

35. FRAME(A341H)::*structured with* INSERTION *field letter i procedure yielding integral field letter r letter e letter p boolean field letter s letter u letter p letter p character field letter m letter a letter r letter k letter e letter r mode.*

Used in transput

mode frame = **struct** (**insertion** *i*, **proc int** *rep*, **bool** *supp*, **char** *marker*)

36. FROBYT(35A)::*from;by;to.*

These appear in loop clauses. It is convenient to group them together in order to define a FROBYT option – all these have the same form.

37. GPATTERN(A341K)::*structured with* INSERTION *field letter i row of procedure yielding integral field letter s letter p letter e letter c mode.*

Used in transput for general patterns.

mode gpattern = **struct** (**insertion** *i*, **flex** [1:0] **proc int** *spec*)

38. HEAD(73B)::PLAIN;PREF;*structured with*;FLEXETY ROWS *of;procedure with;union of;void.*

Includes all things that can start a mode. Comparison of modes then involves checking HEADs and then parts remaining are TAILETYs.

Note PREF includes *procedure yielding.*

39. INDICATOR(48A)::*identifier;mode indication;operator.*

Includes all the symbols used to indicate modes, operators, mode identifiers, label identifiers, etc. This metanotion is used in relating defining and applied occurrences.

40. INSERTION(A41E)::*row of structured with procedure yielding integral field letter r letter e letter p union of row of character character mode field letter s letter a mode.*

    Used in transput.

    mode insertion = flex [1:0] **struct (proc int** *rep*, **union (string, char)** *sa*))

41. INTREAL(12C)::SIZETY *integral*;SIZETY *real*.

    Includes all the modes of integers and reals, of any allowable length.

42. LAB(123K)::*label* TAG.

    Includes QUALITY TAXs for label identifiers. The appearance of '*label l*' indicates that *l* appears as a label in a particular program. Used in the syntax of NESTs and LAYERs. LAB denotes *label*.

43. LABS(123J)::LAB;LABS LAB.

    Includes several (one or more) labels.

44. LABSETY(123I)::LABS;EMPTY.

    Indicates an arbitrary number of label declarations (more precisely n labels where $n \geqslant 0$).

45. LAYER(123B)::*new* DECSETY LABSETY.

    A LAYER is a new sequence of (possibly no) indicator and label declarations. LAYER corresponds to a locale or block. DECSETY appears before LABSETY since declarations must precede labels in reaches.

46. LEAP(44B)::*local*;*heap*;*primal*.

    Encompasses all kinds of generators. (primal generators are used only in the standard prelude).

47. LENGTH(65D)::*letter l letter o letter n letter g*.

    Denotes *long* – see note 1 at the end of this appendix

48. LENGTHETY(65F)::LENGTH LENGTHETY;EMPTY.

    Encompasses all possible sequences of *long* – see note 1 at the end of this appendix.

49. LETTER(942B)::*letter* ABC;*letter aleph*;*style* TALLY *letter* ABC.

    Encompasses all small letters, the unprintable *letter aleph symbol* used in **bits** and **bytes** together with any other style of small letter.

50. LONGSETY(12E)::*long* LONGSETY;EMPTY.

    Encompasses any sequence (including a null sequence) of *longs*. In a program these would be underlined.

    Encompasses any sequence (including a null sequence) of *longs*. In a program these would be underlined.

51. MARK(A341M)::*sign*;*point*;*exponent*;*complex*;*boolean*.

    All frames included under MARK cannot be replicated; a replicator of 1 is always assumed.

52. MEEK(61C)::*unchanged from*;*dereferenced to*;*deprocedured to*.

    Encompasses coercions available in a meek syntactic position.

53. MODE(12A)::PLAIN;STOWED;REF *to* MODE;PROCEDURE; UNITED;MU *definition of* MODE;MU *application*.

Encompasses all modes except **void**. Note that identity declarations do not exist for mode **void** and parameters cannot be of mode **void**.

54. MODINE(44A)::MODE;*routine*.

 Identity declarations come in two kinds. There are MODE identity declarations and routine identity declarations. MODINE is used to cover both and it is used in that part of the syntax which is essentially common to both kinds of identity declaration.

55. MOID(12R)::MODE;*void*.

 Encompasses all possible modes.

56. MOIDS(46C)::MOID;MOIDS MOID.

 The plural of MOID, the set of all possible modes.

57. MOIDSETY(47C)::MOIDS;EMPTY.

 Zero or more MOIDs.

58. MONADIC(542B)::*priority iii iii iii i*.

 Gives priority that one can associate with monadic operator. Formulae have priorities associated with them and the above is the priority of a formula which is of the form monadic operator followed by operand.

59. MONAD(942H)::*or;and;ampersand;differs from; is at most;is at least; over;percent;window;floor;ceiling;plus i times;not;tilde;down; up;plus; minus;style* TALLY monad.

 Includes all operators that can appear as monadic operators or as dyadic operators. See note 3 of this appendix.

60. MONO(123G)::*procedure with* PARAMETER *yielding* MOID.

 Encompasses the modes associated with monadic operators. The operand can have any of the modes PARAMETER and the result can be of any mode MOID. To be precise monadic operators possess routines of mode MONO.

61. MOOD(12U)::PLAIN;STOWED;*reference to* MODE;PROCEDURE;*void*.

 United modes are formed by combining several modes. Each constituent mode must be from the set MOOD.

62. MOODS(12T)::MOOD;MOODS MOOD.

 A united mode has the form
 union of MOODS *mode*.
 See under MOOD.

63. MOODSETY(47B)::MOODS;EMPTY.

 Denotes several (two or more) of the modes that go to make up a united mode. See under MOOD.

64. MORF(61F)::NEST *selection*;NEST *slice*;NEST *routine text*;NEST *ADIC formula*;NEST *call*;NEST *applied identifier with TAG*.

 The set of FORMs is divided into MORFs and COMORFs. The division arises because with FORMs yielding a result of mode NONPROC voiding can be done by deproceduring or voiding. See note 2 of this Appendix.

65. MU(12V)::*mu*TALLY.

 Denotes *mui*, *muii*, and so on. Used in definition of recursive modes. A

recursive mode might begin *mui definition of* .... or *muii definition of* ..... etc. When this mode is used later in the definition (as it must be since the definition is recursive) it appears as *mui application of* ..... or *muii application of* ...., etc. TALLY is used to distinguish different modes.

66. NEST(123A)::LAYER;NEST LAYER.

 A nest is a sequence of LAYERs and corresponds to an environ. Each LAYER corresponds to a new reach and hence a NEST contains the information in different (nested) reaches.

67. NOMAD(942I)::*is less than*; *is greater than*;*divided by*; *equals*; *times*;*asterisk*.

 Includes those operators which can appear as dyadic operators but not as monadic operators. See note 3 of this appendix.

68. NONPREF(71B)::PLAIN;STOWED;*procedure with* PARAMETERS yielding MOID;UNITED;*void*.

 Used in the definition of the predicate *deprefs to firm* to encompass all modes for which result of the predicate is always *false*.

69. NONPROC(67A)::PLAIN;STOWED;REF *to* NONPROC;*procedure with* PARAMETERS *yielding* MOID;UNITED.

 MORFs yielding a result of mode NONPROC and having to be coerced to mode **void** are voided, not deprocedured. This indicates the importance of the NONPROC modes. See note 2 of this appendix.

70. NONSTOWED(47A)::PLAIN;REF *to* MODE;PROCEDURE;UNITED;*void*.

 This encompasses all modes which can never be deflexed. It includes all modes which are not structures or multiple values.

71. NOTETY(13C)::NOTION;EMPTY.

 A (possibly empty) sequence of small letters.

72. NOTION(13A)::ALPHA;NOTION ALPHA.

 Denotes any sequence of small letters. Used especially in the definition of predicates though also elsewhere. Thus NOTION could be replaced by, for example,

 (i) *real*

 (ii) *procedure with real parameter yielding integer* etc.

73. NUMERAL(810B)::*fixed point numeral*;*variable point numeral*; *floating point numeral*.

 Encompasses denotations of integers or reals but does not include SIZE symbols.

74. PACK(31B)::STYLE *pack*.

 NOTION PACK indicates that the particular NOTION is surrounded by brackets. The brackets can be of any of the kinds included under STYLE.

75. PARAMETER(12Q)::MODE *parameter*.

 Encompasses all modes that a parameter may have (**void** is not admitted).

76. PARAMETERS(12P)::PARAMETER;PARAMETERS PARAMETER.

Used (indirectly) in specifying PROCEDURE – which encompasses the modes of all procedures. PARAMETER encompasses modes that a parameter may possess.

77. PARAMETY(12O):: *with* PARAMETERS;EMPTY.

Used in specifying PROCEDURE which encompasses modes of all procedures. Includes procedures with parameters and procedures which do not have parameters.

78. PART(73E)::FIELD;PARAMETER.

Encompasses fields in selectors and parameters in procedures. Note that they are very similar provided one strips of **struct** ( ) from the structure and the (and) which surround parameters. PART is used in determining the equivalence of two modes.

79. PARTS(73D)::PART;PARTS PART.

Encompasses one or more PARTs. See under PART.

80. PATTERN(A341G)::*structured with integral field letter t*
letter y letter p letter e
row of FRAME *field*
letter f letter r letter a letter m letter e letter s mode.
Used in transput
mode pattern = **struct** (**int** *type*, **flex** [1:0] **frame** *frames*).

81. PICTURE(A341F)::*structured with union of* PATTERN CPATTERN FPATTERN GPATTERN *void mode field letter p* INSERTION *field letter i mode.*
Used in transput
mode picture = **struct** (**union** (**pattern**, **cpattern**, **fpattern**, **gpattern**, **void**) *p*,**insertion** *i*)

82. PIECE(A341B)::*structured with integral field letter c letter p integral field letter c letter o letter u letter n letter t integral field letter b letter p row of* COLLECTION *field letter c mode.*
Used in transput.
mode piece = **struct** (**int** *cp*, *count*, *bp*, **flex** [1:0] **collection** *c*)

83. PLAIN(12B)::INTREAL;*boolean*;*character*.

Encompasses all the plain values i.e. arithmetic values (of any size), booleans and characters.

84. PRAGMENT(92A)::*pragmat*;*comment*.

The meaning of any program is unaffected by either comments or pragmats. Comments should be entirely ignored by the implementation whereas pragmats should convey some information to the implementation.

85. PRAM(45A)::DUO;MONO.

Encompasses modes of routines associated with operators, i.e. procedures with one or two parameters yielding results of a particular kind.

86. PREF(71A)::*procedure yielding*;REF *to*.

Used in testing well-formedness of modes.

The heads included under PREF are those which supply a *yin* only i.e. they are the *yin*-heads which are not *yang*-heads.

87. PREFSETY(71C)::PREF PREFSETY;EMPTY.

Meekly related modes must be of the form MODE and PREFSETY MODE. PREFSETY therefore includes any initial sequence of **proc** and **ref**.

88. PRIMARY(5D)::*slice coercee;call coercee;cast coercee;denoter coercee;format text coercee;applied identifier with* TAG *coercee*; ENCLOSED *clause.*

Encompasses all constituents that are PRIMARY – used for the purpose of implied bracketing.

89. PRIO(123F)::*i;ii;. . . . . ;iii iii iii.*

Gives all possible priorities of dyadic operators i.e. *i, ii, ... , iii iii iii.*

90. PROCEDURE(12N)::*procedure* PARAMETY *yielding* MOID.

Encompasses the modes of all routines.

91. PROP(48E)::DEC;LAB;FIELD.

Each indicator (mode or label identifier, mode indication or operator indication) possesses a property i.e. it identifies either

a declaration of some kind (excluding label) – hence DEC;
a label – hence LAB;
or a field of a structure – hence FIELD.

PROP therefore encompasses the QUALITY TAX for each such object.

92. PROPS(48D)::PROP;PROPS PROP.

One or more properties – see under PROP.

93. PROPSETY(48C)::PROPS;EMPTY.

A sequence of zero or more properties. See under PROP.

94. QUALITY(48F)::MODE;MOID TALLY;DYADIC;*label*;MODE *field.*

Notice that PROP is defined in terms of DEC, LAB and FIELD. When expanded these give MODE TAG; *priority* PRIO TAD; MOID TALLY TAB; DUO TAD; MONO TAM; *label* TAG; MODE *field* TAG. Each indicator i.e. TAX (except for operators and these are exceptions) has a quality attached to it. The qualities are encompassed by QUALITY. The QUALITY TAX associated with an item is then used to relate defining and applied occurrences of objects.

95. RADIX(82A)::*radix two;radix four;radix eight;radix sixteen.*

Bits denotations can be expressed in radix 2,4,8, or 16. RADIX encompasses all these possibilities.

96. REF(12M)::*reference;transient reference.*

Includes references or transient references.

97. REFETY(531A)::REF *to*;EMPTY.

Includes *reference to*, *transient reference to* or EMPTY. A selection will always deliver a result of mode REFETY MODE where MODE is the mode of a particular field of the structure.

98. REFLEXETY(531B)::REF *to*;REF *to flexible*;EMPTY.

One can subscript or select from an object of mode beginning with REFLEXETY.

99. ROWS(12L)::*row*;ROWS *row*.

Denotes *row*, *row row*, *row row row* etc. These characterise, respectively, one dimensional, two dimensional, three dimensional, etc. multiple values.

100. ROWSETY(532A)::ROWS;EMPTY.

Includes any number of rows (including none). See ROWS.

Note that subscripting a multiple value can leave a result where mode contains zero rows, or several rows.

101. SAFE(73A)::*safe*;MU *has* MODE SAFE;*yin* SAFE;*yang* SAFE; *remember* MOID1 MOID2 SAFE.

Used in determining whether two modes are equivalent. SAFE acts as a stack onto which one pushes several quantities

yin denotes one has encountered a *reference to*, *a transient reference to* or a procedure;

yang indicates one has encountered a structure or a procedure with parameters;

remember MOID1 MOID2 indicates that in the course of comparing two modes the modes MOID1 and MOID2 have been tested for equivalence;

MU *has safe* MODE indicates that a recursive mode namely MU *definition of* MODE has been encountered.

102. SECONDARY(5C)::LEAP *generator coercee*;*selection coercee*; PRIMARY.

Encompasses all the constructions forming a SECONDARY; used for purposes of implicit bracketing.

103. SHORTH(65E)::*letter s letter h letter o letter r letter t*.

Denotes *short* – see note 1 at the end of this appendix.

104. SHORTHETY(65G)::SHORTH SHORTHETY;EMPTY.

Encompasses all allowable sequence of *short* – see note 1 at the end of this appendix.

105. SHORTSETY(12F)::*short* SHORTSETY;EMPTY.

Encompasses any sequence (including a null sequence) of *shorts*. In a program these will be underlined since they are part of a mode. See note 1 at the end of this appendix.

106. SITHETY(65C)::LENGTH LENGTHETY;SHORTH SHORTHETY; EMPTY.

Encompasses all sequences of *long* or sequences of *short* which appear as parts of selectors (see note 1).

107. SIZE(810A)::*long*;*short*.

Includes the two ways of obtaining more or less precision in integers and reals.

108. SIZETY(12D)::*long* LONGSETY;*short* SHORTSETY;EMPTY.

Includes all the sizes of integers and reals. Thus one can use first modes

int and **real** by making use of EMPTY or one can have any sequence of *longs* or any sequence of *shorts*. Note that *longs* and *shorts* cannot be mixed.

109. SOFT(61D)::*unchanged from*;*softly deprocedured to*.
 Encompasses coercions available in a soft syntactic position i.e. leaving the mode of the object unchanged or deproceduring.

110. SOID(31A)::SORT MOID.
 Encompasses all possible strengths of position and all possible modes.

111. SOME(122B)::SORT MOID NEST.
 SOME encompasses all combinations of SORT, MOID and NEST.

112. SORT(122C)::*strong*;*firm*;*meek*;*weak*;*soft*.
 Includes strengths of all syntactic positions.

113. STANDARD(942E)::*integral*;*real*;*boolean*;*character*;*format*;*void*;*complex*; *bits*;*bytes*;*string*;*sema*;*file*;*channel*.
 Includes all modes in the standard environment. These cannot be altered by mode declarations.

114. STOP(A1B)::*label letter s letter t letter o letter p*.
 The particular postlude of each particular program is executed just before the program ends. Its instructions to lock the standard files are preceded by the label *stop*.

115. STOWED(12H)::*structured with* FIELDS *mode*;FLEXETY ROWS *of* MODE.
 Encompasses the modes of all kinds of structures or multiple values. Only a proper subset of the modes included in the set STOWED can be deflexed. Modes not in the set STOWED cannot be deflexed.

116. STRONG(61A)::FIRM;*widened to*;*rowed to*;*voided to*.
 Encompasses all coercions available in a strong syntactic position. Includes these mentioned explicitly above together with uniting, dereferencing and deproceduring.

117. STYLE(133A)::*brief*;*bold*;*style* TALLY.
 Used to group together symbols of a similar nature. For instance
 (i) STYLE *comment symbol* groups the different kinds of symbols that can be used to surround comments. There are *brief comment symbols* and *bold comment symbols*. The *style* TALLY allows the implementor to include other comment symbols if he wishes.
 (ii) There are STYLE opening brackets and STYLE closing brackets. The STYLE and consistent substitution imply that
 begin and **end** must match
 and (and) match

118. TAB(942D)::*bold* TAG;SIZETY STANDARD.
 Includes identifiers which are underlined or are within quotes, i.e. are made bold in some way, together with any of the standard modes preceded by any number of **longs** or any number of **shorts** but not a mixture. TABs are used for specifying modes.

119. TAD(942F)::*bold* TAG;DYAD BECOMESETY;DYAD *cum* NOMAD BECOMESETY.

 Includes all the sequences of symbols that can be used to identify dyadic operators. (See note 3).

120. TAG(942A)::LETTER;TAG LETTER;TAG DIGIT.

 Any sequence of letters and/or digits starting with a letter – as used to form identifiers, selectors, etc.

121. TAILETY(73C)::MOID;FIELDS *mode*;PARAMETERS *yielding* MOID;MOODS *mode*;EMPTY.

 Includes all parts of mode remaining when the HEAD is taken from the mode. Thus determining whether two modes are identical is a matter of comparing the HEADs and then the TAILETYs.

122. TALLETY(542D)::TALLY:EMPTY.

 Used in counting, used in particular in definition of a formula to determine implied bracketing. Left hand of DYADIC formula has priority DYADIC TALLETY and right hand operand has priority DYADIC TALLY. This latter priority is strictly greater than that represented by DYADIC. Thus

$$a - b - c$$

has an implied bracketing

$$(a - b) - c$$

123. TALLY(12W)::*i*;TALLY *i*.

 Used in counting. See under TALLETY. TALLY is used also in

 (i) making sure one style of opening brackets matches another style. This can be done by *style* TALLY.

 (ii) distinguishing one recursive mode definition from another. This is done by using *mui* TALLY and TALLY is replaced by different values for the different modes.

124. TAM(942K)::*bold* TAG;MONAD BECOMESETY;MONAD *cum* NOMAD BECOMESETY.

 Includes all possible sequences of symbols used to denote monadic operators (see note 3).

125. TAO(45B)::TAD;TAM.

 Includes the sequences of symbols used to identify operators. TAD takes care of dyadic operators and TAM takes care of monadic operators. (See note 3).

126. TAX(48G)::TAG;TAB;TAD;TAM.

 Includes all sequences of symbols used to identify modes (TABs), monadic operators (TAMs), dyadic operators (TADs) and selectors, identifiers or labels (TAGs).

127. TERTIARY(5B)::ADIC *formula coercee*;*nihil*;SECONDARY.

 Encompasses all the constructions that make up a TERTIARY; used for purposes of implied bracketing.

128. THING(13D)::NOTION;(NOTETY1)NOTETY2;THING(NOTETY1)

NOTETY2.

Sequence of small letters with, perhaps, one or more sets of matching bracket pairs (and) inserted. (and) cannot be nested at all.

129. TYPE(A341P)::*integral*;*real*;*boolean*;*complex*;*string*;*bits*;*integral choice*; *boolean choice*;*format*;*general*.

   In formatted transput each pattern has a type associated with it. TYPE includes the types of all such patterns.

130. UNIT(5A)::*assignation coercee*;*identity relation coercee*;*routine text coercee*;*jump*;*skip*;TERTIARY.

   Encompasses all the constructions that can be units (unitary clauses).

131. UNITED(12S)::*union of* MOODs *mode*.

   Encompasses all modes involving unions. Unions are formed by combining only those modes included under MOOD.

132. UNSUPPRESSETY(A3410)::*unsuppressible*;EMPTY.

   Frames included under MARK and COMARK can be suppressed. RADIX frames cannot be suppressed. UNSUPPRESSETY includes both possibilities.

133. VICTAL(46A)::VIRACT;*formal*.

   Includes all kinds of declarers: virtual, actual and formal.

134. VIRACT(46B)::*virtual*;*actual*.

   Used in connection with declarers.

135. WHETHER(13E)::*where*;*unless*.

   Used in the definition of predicates.

Note 1

Mode **bits** is a structure with selector *aleph symbol* (a symbol with no hardware representation and consequently selection cannot take place). Mode **long bits** has selector *long aleph symbol*, **short bits** has selector *short aleph symbol*, etc. and none of these symbols has a hardware representation. **long long bits** has selector *long long aleph symbol* and so on. Thus *long* and *short* are used to provide different identifiers for the selectors of different lengths of bits allowed. Note that these different selectors imply that the modes of different sizes of **bits** are different.

In the syntax *long*, part of a selector, appears as *letter l letter o letter n letter g*. Within modes it appears in its abbreviated form, i.e. as *long*. Similarly for *short*, *long long*, *short*, *short*, etc.

Note 2

See discussion in notes on FORMs, MORFs, COMORFs and NONPROCs.

Note 3

To explain the grouping of DYAD into MONAD and NOMAD. DYAD includes all the operator symbols. Some of these can be used monadically and they appear in the set MONAD. Now the symbols included in DYAD can be

combined in pairs to produce operator symbols. Ambiguities will arise unless certain precautions are taken. Two symbols from MONAD cannot be combined to produce a new monadic operator. For one must be able to tell whether such a symbol would represent a single application of the new operator or a double application of the old operator.

For similar reasons a DYAD cannot be followed by a MONAD since ambiguity would result. Is the new symbol an application of the old operator with a right hand operand operated on by the monadic operator?

These observations explain the appearance of DYAD *cum* NOMAD in the metarule for TAD and MONAD *cum* NOMAD in the rule for TAM.

APPENDIX B THE PREDICATES

Included here is a list of predicates used in the formal definition of ALGOL 68 together with some explanation as to their meaning.

1. *and* (1.3.1c,e)

   This is equivalent to boolean **and**. Thus to ask

   where THING1 *and* THING2

   is to insist that both *where* THING1 disappears

   and *where* THING2 disappears

2. *balances* (3.2.1f,g)

   balances applies to both SORTs and MOIDs (when applied to SOIDs its effect is just the combined effect on the SORT and MOID).

   (i) SORT balances SORT1 and SORT2 if one of the latter two is strong and the other is SORT

   (ii) MOID balances MOID1 and MOID2 provided
   all three are the same
   all three are the same except that either (but not both) MOID1 or MOID2 has a transient missing.

   Note that this predicate does not give all the details about how two modes should be balanced.

3. *begins with* (1.3.1h,i,j)

   This predicate asks whether one sequence of small letters begins another. Thus

   where (*aaa*) *begins with* (*aa*) disappears and
   where (*abc*) *begins with* (*abc*) also disappears.

4. *coincides with* (1.3.1k,l)

   Asks whether two small letters coincide with each other in a particular notion.

   where (*a*) *coincides with* (*a*) *in* (NOTION)

   always disappears no matter what the NOTION or the particular small letter *a*

   unless (*a*) *coincides with* (*b*) *in* NOTION always disappears

   if (NOTION) contains a sequence *a* . . . . *b* or *b* . . . . *a*. Here *a* and *b* are used to denote any small letters; they may even denote the same letter. Thus

   unless (*a*) *coincides with* (*a*) *in* (*a*) delivers *where false*
   unless (*a*) *coincides with* (*a*) *in* (*aa*) delivers *where true*.

5. *contains* (1.3.1m,n)

Indicates if a sequence of small letters contains another sequence

where (*abc*) *contains* (*bc*) delivers *where true*

where (*abc*) *contains* (*ac*) delivers *where false*.

6. *counts* (4.3.1c,d)

Used to compare digit symbols and priority levels. Gives EMPTY, i.e. disappears, if the digit and priority levels are "equal" in usual sense; e.g.

where digit seven counts prio iii iii i disappears

7. *deflexes to* (4.7.1a,b,c,d,e).

Certain modes have to be deflexed (i.e. flexible is removed) so that values can be assigned to them. Values cannot be flexible. This predicate indicates whether it is possible. (Taken from R2.1.3.6b)

> "The deflexing process obtains MOID2 by removing all flexibles contained at positions in MOID1 where they are not also contained in any REF *to* MOID3. Thus flexible can be removed from the various fields of a structure or from a multiple value (but not references to multiple values or references to structures)."

deflexes to is used to obtain the nonflexible mode corresponding to a possibly flexible STOWED mode.

8. *deprefs to firm* (7.1.1n)

In determining whether two modes are firmly related it is necessary to decide if one can go from one mode to another by means of dereferencing, deproceduring and uniting. The predicate *deprefs to firm* takes care of dereferencing and deproceduring. See also the predicate *is firm*.

9. *develops from* (7.3.1c)

In determining the equivalence of two modes (see the predicate equivalent) the syntax trees corresponding to these modes are constructed (or developed) until they are shown to be different or until no differences can be found. The predicate *develops from* does just this. Used to either

(a) break mode into its head and tail; then a test can be performed on the heads and tails and an appropriate *yin* or *yang* placed in the SAFE

(b) replace applied occurrences of modes by their expanded form

(c) in the case of recursive mode definitions it strips off the MU *definition of*, places MU *has* in the SAFE and proceeds as in (a), (b) and (c) with MU *definition of* MODE replaced by just MODE.

10. *equivalent* (7.3.1)

MOID1 is equivalent to MOID2 if MOID1 and MOID2 provide possibly different spellings of one same mode. Thus

(i) **union (real, char)** and **union (char, real)** are equivalent

(ii) **a** and **b** are equivalent spellings given that

mode a = struct (ref a *b*)

mode b = struct (ref struct (ref b *b*) *b*)

(iii) any spelling of a mode is always equivalent to itself.

11. *false* (1.3.1b)

unless false disappears. Therefore in considering predicates starting with

unless one tries to produce alternatives ending in *unless false*.

12. *firmly related* (7.1.1k)

Two modes are firmly related if they are both coercible from one same mode, e.g. **int** and **union (ref ref int, char)** are firmly related, the common mode being **ref ref int**.

This predicate is important in determining whether two PROPs are independent. In particular it enables one to determine if two dyadic operator declarations are independent; it also determines if two modes can be components of the same united mode.

See also the predicate *is firm*.

13. *identified in* (7.2.1a)

Used in relating applied and defining occurrences of indicators; determines whether a given indicator has been declared in any enclosing LAYER, i.e. within the current environment.

14. *incestuous* (4.7.1f)

A component mode of a union may not be firmly related to one of the other component modes or to a union of component modes. The predicate *incestuous* determines if this is the case. Thus one set of modes, set 1, is incestuous with another set, set 2, if set 2 contains a mode which can be firmly related to one of the other component modes in set 1 and set 2 or to a union of these component modes.

15. *independent* (7.1.1a,b,c,d)

Two properties are independent if they can exist within the same layer (i.e. within same reach). This predicate determines independence. The properties corresponding to **real** *x* and **int** *x* for example are not independent.

16. *is* (1.3.1g)

Determines whether 2 strings of small letters are identical. Includes possibility of strings being empty.

where (aaa) is (aaaa) produces *where false*
where ( ) is ( ) produces *where true*.

17. *is derived from* (5.3.1.1b.c)

One can subscript or select from an object of mode beginning REFLEXETY. The result produced is of mode beginning REFETY where REFETY is derived from REFLEXETY. The table below gives the different possibilities

REFETY	(*is derived from*)	*REFLEXETY*
EMPTY		EMPTY
reference to		*reference to*
transient reference to		*transient reference to*
transient reference to		*reference to flexible*
transient reference to		*transient reference to flexible*

18. *is firm* (7.1.1l and m)

In determining whether two modes are firmly related it is necessary to

decide if one can go from one mode to another by means of dereferencing, deproceduring and uniting. The predicate *is firm* essentially breaks united modes into component modes and performs the individual tests. See also the predicate *deprefs to firm*.

19. *like* (5.4.1.1c,d)

When a procedure is called the modes of the actual parameters must be *like* the modes of the corresponding actual parameters.

20. *meekly related* (not in new report)

Two modes are meekly related if they are both meekly coercible from one same mode.

21. *may follow* (3.4.1m)

Used to ask if one type of choice clause can follow another. If first part is choice using integer or boolean then the part following the **elif** or **ouse** must also be integer or boolean respectively. If it is a choice using **union** then the second part must also be a choice using a **union**.

22. *number equals* (7.3.1o,p)

To test if there are the same number of objects in one set as in another. In determining the equivalence of two united modes the number of constituent modes must be the same in each case.

23. *or* (1.3.1d,f)

Equivalent to boolean **or**. Thus to ask

where THING1 *or* THING2

is to ask if just one of

where THING1 produces *where true*

or

where THING2 produces *where true*.

24. *ravels to* (4.7.1g)

A set of MOODs and UNITEDs may be ravelled by replacing all UNITEDs by their constituent MOODS, i.e. reduces a given set of modes into an equivalent set from which all (intermediate) 'union of's have been deleted.

25. *related* (7.1.1e,f,g,h,i,j)

Used in testing independence of PROPs – a declaration is legal only if it is independent of other declarations in that reach. Declarations involving operators often use the same indication. The predicate *related* tells whether such declarations are allowed. In particular

(i) priority declarations are independent of operator declarations and vice versa

(ii) monadic operator declarations are independent of dyadic operator declarations and vice versa

(iii) two dyadic operator declarations are independent unless the corresponding operands are firmly related.

related compares QUALITYs of two operators with the same TAX.

26. *resides in* (7.2.1b,c)

    Used in relating defining and applied occurrences of indicators; determines if a given indicator has been declared in a particular reach.

27. *shields* (7.4.1a,b,c,d)

    Used in testing the well-formedness of modes. For recursive modes there must be a SAFE. For a mode to be well-formed a *yin* and a *yang* must be placed in the SAFE. *shields* describes those parts of a mode placing a *yin* or *yang* in the SAFE.

28. *subset of* (7.3.1.1,m,n)

    Determines if one set of objects is a subset of another. In deciding if two united modes are equivalent it is necessary to decide if the set of constituent modes of one union is a subset of the constituent modes of the other and vice versa.

    Also one united mode can be coerced to another united mode only if the modes contained in the first union are a subset of the modes contained in the second union.

29. *true* (1.3.1a)

    where true delivers EMPTY. Therefore in considering predicates starting with *where* one tries to produce alternatives ending in *where true*.

30. *unites to* (6.4.1b)

    Predicate deciding if one mode can be united to another mode.

REFERENCES

1. BAUER, H., BECKER, S., GRAHAM, S. and SATTERTHWAITE, E. *ALGOL W Language Description*, Computer Science Department, Stanford University, 1968.
2. EVE, J. *ALCOL W Programming Manual*, from University of Newcastle-upon-Tyne.
3. KNUTH, D. E. The remaining trouble spots in ALGOL 60, *Comm. ACM 10*, **10**, 1967, p 611.
4. LEDGARD, H. F. Production Systems: or Can We Do Better Than BNF? *Comm. ACM 17*, **2**, February 1974, pp 94–102.
5. LUCAS, P. *et al. Method and notation for the formal definition on programming languages*, TR 25.087, IBM Lab. Vienna, 1968.
6. LUCAS, P. and WALK, K. On the formal definition of PL/I, *Annual Review in Automatic Programming* **6**,**3**, 1969.
7. McCARTHY, J. *A basis for a mathematical science of computation.* In *Formal Programming Languages*, Braffort and Hirschberg (Eds), North Holland Publ. Co., Amsterdam, 1963.
8. McCARTHY, J. *A formal description of a subset of ALGOL*. In *Formal Language Description Languages for Computer Programming*, T.B. Steel, Jr (Ed), North Holland
9. McCARTHY, J. *The Lisp 1.5 Programming Manual* M.I.T. Press, Cambridge, Mass., 1965.
10. McCARTHY, J. *Towards a mathematical science of computation.* In *Proc. IFIP Cong.* 1962, North Holland Pub. Co., Amsterdam, 1963.
11. McCARTHY, J. and PAINTER, J. *Correctness of a compiler for arithmetic expressions.* In *Proc. Symposium on Appl. Math.*, Vol. **19**, Amer. Math. Soc., 1967.
12. NAUR, P. Revised report on the algorithmic language ALGOL 60. *Comm. ACM 6*, **1** Jan. 1963, pp 1–17.
13. SINTZOFF, M. On the revised ALGOL 68 Report. *ALGOL Bulletin No. 36*, Nov. 1973, AB 36.4.3, pp 28–39.
14. van WIJNGAARDEN, A. *et al.* Report on the Algorithmic Language ALGOL 68. *Numerische Mathematik 14*, **12**, 1969, pp 84–218.
15. van WIJNGAARDEN, A. *et al.* Revised Report on the Algorithmic Language ALGOL 68, *Acta Informatica*, Vol. **5**, Fasc 1–3, 1975, pp 1–236.
16. WALK, K. *et al. Abstract syntax and interpretation of PL/I, Version III.* TR 25.098, IBM Lab Vienna, April 1969.
17. WEGNER, P. The Vienna Definition Language, *ACM Computing Surveys* **4**, 1 March 1972, pp 5–63.
18. WIRTH, N., and HOARE, C. A. R. A Contribution to the Development of ALGOL, *Comm. ACM*, Vol. **9**, No. 6, 1966.
19. *The New Encyclopaedia Britannica* (Micropaedia Volume X), 15th edition, 1974, Encyclopaedia Britannica, Inc., William Benton Publishers.

ERRATUM

Under "Contents of previous Volumes"

The following articles did not appear in Volume 6.

The Language of Computers — Lord Bowden

A Compiler Generating System — K. Fujino